VIETNAM NATIONAL UNIVERSITY OF HOCHIMINH CITY
THE INTERNATIONAL UNIVERSITY
SCHOOL OF COMPUTER SCIENCE AND ENGINEERING

A MODERN APPROACH FOR SOFTWARE-BASED THESIS MANAGEMENT: PLATFORM FOR SEMI-AUTOMATIC DEPLOYMENT AND MONITORING WITH KUBERNETES

By
Nguyễn Lê Nguyễn

A thesis submitted to the School of Computer Science and Engineering
in partial fulfillment of the requirements for the degree of
Bachelor of Information Technology/Computer Science/Computer Engineering

Ho Chi Minh City, Vietnam
2021

A MODERN APPROACH FOR SOFTWARE-BASED THESIS MANAGEMENT: PLATFORM FOR SEMI-AUTOMATIC DEPLOYMENT AND MONITORING WITH KUBERNETES

APPROVED BY:

________________________________ ,
Vo Thi Luu Phuong, Assoc. Prof., Chair (*Example*)

____
Huynh Kha Tu, PhD

____
Le Thanh Son, MSc

____
Tran Thanh Tung, PhD

____
Le Duy Tan, PhD

THESIS COMMITTEE 2

ACKNOWLEGMENTS

It is with deep gratitude and appreciation that I acknowledge the professional guidance of Dr. Tran Thanh Tung. His understanding and support has helped me tremendously in achieve my goal with this thesis. My gratitude also goes to all members of the CVIP laboratory, who have helped me build my technical prowess and self-learning skill. Their knowledge and guidance over the years has assisted me greatly in achieving a better version of myself and I am deeply grateful for that. I am also grateful to the faculty of the School of Computer Science of the International University, especially Dr. Ly Tu Nga for her continuous support on the academic affairs side since the start of my study here at HCMIU. Gratitude is also expressed to the members of my reading and examination committee for spending their valuable time.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ABSTRACT

As the market for software gets larger, every moment there are more and more university theses developed about and along with software or software systems. However, current thesis management systems on the market do not support the maintenance of old thesis software, only the thesis information. Old thesis software is abandon with little to no way of recovering them unless the original owner is present. This is a sizeable problem and is creating massive losses in intellectual property and creativity. In this thesis, I investigate the feasibility of modern containerization technologies and their possible application in maintaining old thesis software to propose a more suitable thesis management system for the modern software world, equipped with Kubernetes and Docker to solve the increasing old thesis software maintenance problem. In this report, a design for the suitable system is drafted and developed along with experiments to demonstrate the capability of Kubernetes in deploying and maintaining old software.

CHAPTER 1

INTRODUCTION

1.1. Motivation

In today's information age, as all workflows and assets are getting digitalized to increase their efficiency, new software gets developed and deployed every passing moment. This significantly drives the rise in market demand for software development in particular, and the growth of the computer science field as a whole. In turn, this pushed the expansion of the Computer Science faculty in most universities and increased the number of software-based theses that result from the graduation of Computer Science students.

The ever-increasing number of software-based theses lead to the ever-rising demand in a proper management system. A proper thesis management system is one that can keep track of information on a student's thesis, as well as being capable of maintaining its functionality long after they graduated. The system may also provide swift recreation of the demonstrative environment for qualifying theses, allowing future review of an old, but working software. As of the writing of this document, there is no prominent solution for such a system, especially in domestic universities. This is a **waste of manpower** and **creativity** of the students, who pour their hearts and souls into developing their hard-earned theses. The call for a suitable solution for such a management system is deafening.

A software capable of maintaining and deploying other software is unsurprisingly going to be quite complex. As with the solution to many complex problems since the dawn of computer science, **abstraction** is the key to success. Abstraction is a very powerful tool in computer science, allowing software to be developed in a human-comprehensible language while being able to run on complex systems made up of merely electrical components. With the rise of **Virtualization** technology, which abstracts away the infrastructure for deploying software, along with the rise of **Containerization** technology, which abtracts away the runtime environment for software execution, a software system capable of managing the maintenance and deployment of other software becomes increasingly feasible.

The state-of-the-art defining tool for containerization technology and one that heavily utilizes virtualization technology is Docker. Docker grants quick deployment of Docker-containerized software on any platform, as long as it supports virtualization. This makes Docker seem related and similar to another popular cross-platform technology, Java. Both technologies allow almost seamless transition for software deployment between multiple platforms using abstraction. However, this is where the similarities end. While Java provides cross-platform capabilities, it is at a lower abstraction level than Docker and thus requires much more manual configuration to setup the runtime environment. Some software, even though written in Java, won't work cross-platforms unless some dependency binaries are rebuilt or externally provided. Docker, on the other hand, abstracts away the entire operating system on which the software is being executed. This eliminates the need for platform-specific binaries and reduces the amount of manual configuration needed to run a particular software, since much of the environment configuration is included the build phase of a software container. With the deployment capability of Docker, redeploying old software on continuously evolving environments suddenly seems realistically achievable. However, Docker is only one part of the puzzle to the solution of a software management system. The other crucial part is the orchestration of Docker containers.

Enter Kubernetes, one of the most popular open-sourced container orchestration technology on the current market. Kubernetes allows the automated deployment, scaling and management of containerized software, and is designed to work in tandem with many container runtimes, especially Docker. Both of these technologies, Docker and Kubernetes, share a symbiotic relationship. While Docker provides the runtime for the containers being orchestrated by Kubernetes, the latter provides automatic configuration and maintenance for the Docker containers. Docker, even with all the former levels of abstraction described, still needs manual configuration for some software-specific runtime environment variables. Kubernetes eliminates this problem entirely, by providing another level of abstraction for containers' configuration, allowing a single definition of the runtime environment configuration to be reused automatically in later deployments. Kubernetes also provides automatic networking between multiple containers, increasing the redeployability of software based on the microservice architecture. This is an essential for the software redeployment capability of any would-be software management system.

With the rising popularity of Containerization and Virtualization technologies like Docker and Kubernetes aiding their continuous development, a proper software management system suddenly becomes technically feasible. With Docker, old software can now be defined in as a blueprint and be deployed onto newer infrastructure with minimal hassle. With Kubernetes, the redeployment of old Docker container can now be automated with predefined configurations, minimizing the amount of manual work required to spin up an old software. This is ideal for the long-overdue thesis management system. Thanks to these new technological advancements, the question for a proper software-based thesis management system capable of rapidly recreating demonstrative environment for old, qualified thesis software, providing future review capability, is no longer the question of how, but a question of when.

1.2. Problem Statement

The aim of this thesis is to create a platform where software-based thesis can be managed and maintained, and where suitable old thesis software can be deployed for future reviews. The targeted system will also be a demonstration for the potential for industry applications of these kinds of software management systems that utilize virtualization and containerization.

1.3. Scope and Objectives

As previously described, containerization and virtualization will be heavily utilized to build such a complex software management system. In addition, the automatic orchestration and management of containerized applications is also imperative. As such, to match these requirements, research, and practice with contemporary technologies such as Docker and Kubernetes must be done. The primary goal of this is to grasp an understanding of these technologies as well as to efficiently utilize them to produce a sufficiently capable software-based thesis management system.

Furthermore, in order for this management platform to be easily accessible and user friendly, a web application will be used to serve as the user's interface with the system. For this web application, to keep things simple and minimize the time resource spent, the web application system will be comprised of a NodeJS backend server which will be providing a ReactJS frontend web application with a GraphQL API to access the thesis information stored in a PostgreSQL database as well as the underlying Kubernetes cluster information.

The main objective of this system is to provide:

- A web-based thesis management system to view submitted thesis information
- A beginner-friendly UI/UX for submitting thesis description and thesis software (minimal knowledge about containerization needed)
- Automatic redeployment of old qualified thesis software onto the system, ready to be view
- Administrative management capability such as approving, deleting submit requests, create, and distribute system resource quota for each thesis software
- A Kubernetes cluster for hosting Docker-containerized thesis deployments.
-

1.4. Assumption and Solution

In the earlier sections, there were a lot of mentions about "qualified theses" suitable for redeployment and future reviews, the criteria for these will be assumed in this section and so will the solution's approach for accomplishing the redeployment of these theses.

The 3 main criteria for a qualifying thesis are:

- Is software-based, meaning the demonstration/experiment section of the thesis is a runnable application
- Must be able to be containerized using Docker
- Has a web-based user interface (in order to be interactable from the web-based management platform)

Some non-qualifying theses include but are not limited to research papers and statistical survey studies. For the qualifying theses, the basic approach for hosting and deploying the software is using Docker and Kubernetes:

- Docker will be used to containerize the qualifying theses' software
- Kubernetes will be used to manage a cluster of Docker containers, deploying the software when requested and terminating them when not in used

CHAPTER 2

LITURATURE REVIEW/RELATED WORK

In order to acquire the knowledge for designing and implementing the complex thesis management system stated in Chapter 1, research and literature review of virtualization, containerization, and container orchestration technologies are required. These mainly include Docker and Kubernetes. Modern web technologies such as GraphQL is also researched in attempt to better optimize the design of the thesis management system. Other related works are also reviewed and differentiated.

2.1. Docker

So, what exactly is Docker? In its own words, as described in the official documentation, "Docker is an open platform for developing, shipping, and running applications" (1).

The Docker platform enables applications to be separated from any operating system or hardware, thus eliminating compatibility, infrastructure, and environment dependant issues. Docker accomplishes this through the use of isolated runtime environments called containers. Each container is an isolated environment, similar to a mini virtual machine, created from an application blueprint called a Docker image. Every Docker image is in turn built from a base image, usually a predefined Docker image of an operating system or an application, along with any commands to build the application on top of the base image through a configuration file called a Dockerfile. The Docker images can then be used by the Docker Engine to create and run Docker containers.

The containerization of applications though Docker containers is what makes Docker a terrific tool for running applications on top of any infrastructure or hardware. However, the same could be said for virtualization technologies that have existed long before Docker, so what makes Docker containers better than just using hypervisors to create new virtual machines? This will be discussed in subsection 2.1.1. As mentioned in the previous chapter, Docker containers are also an integral part of container orchestration system such as Kubernetes, this will be elaborated in section 2.2.

2.1.1. Containerization vs Virtualization

Containerization is a form of virtualization in which applications are executed in isolated containers, using isolated resource spaces shared by the host machine while sharing the same operating system as the host. General virtualization on the other hand, usually uses hypervisors, or Virtual Machine Monitors (VMMs), to create and run virtual machines (VMs), which also use isolated resource spaces shared from the host machine but on top of their own instances of guest operating systems (2).

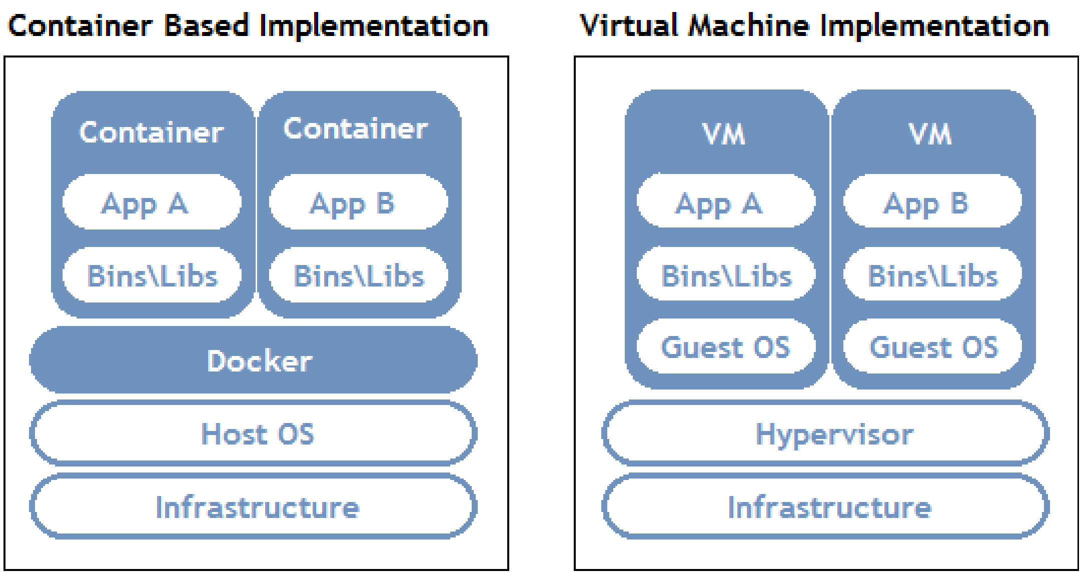


Figure 1. Containers vs Virtual Machines

The old approach of full-fledge virtual machines requires a lot more resources to run, since each VM has its own OS and that requires a lot of processing power. Docker's containerization approach, however, abstract away the operating system and allow containers to communicate directly with the host machine's kernel. This results in containers being much more lightweight than virtual machines can ever be. Containers use much less resource in processing power, memory and time to run the applications.

Even with all its benefits, Docker is not without its drawbacks. While hypervisors can and have always been used on many different operating systems to run virtual machines on other operating systems, Docker only supports the Linux kernel. In addition, virtual machines using hypervisors provide a higher level of data security, due to the fact that there are at least 2 levels of OS security. Docker, on the other hand, only provides as much security as the kernel of the Linux distribution does since Docker containers communicate directly with the host OS's kernel. Even so, Docker does provide support for running Linux and Windows applications on not only Linux, but also Windows and MacOS (3). Although running natively on Linux will usuall yield the best Docker experience.

2.1.2. Docker architecture

As for its architecture, similar to many modern applications, Docker use a client-server architecture. Docker provides its user with a Docker *client*, usually as a command-line application, to interact with the server, or the Docker *daemon*, which does most of the work of building, running, and distributing Docker containers. Along with the basic Docker *client*, there is another client called *Docker Compose*, which provides support for applications that require running a set of containers instead of just one. These Docker components communicate with each other through a REST API, usually over UNIX sockets or through a network interface (4)

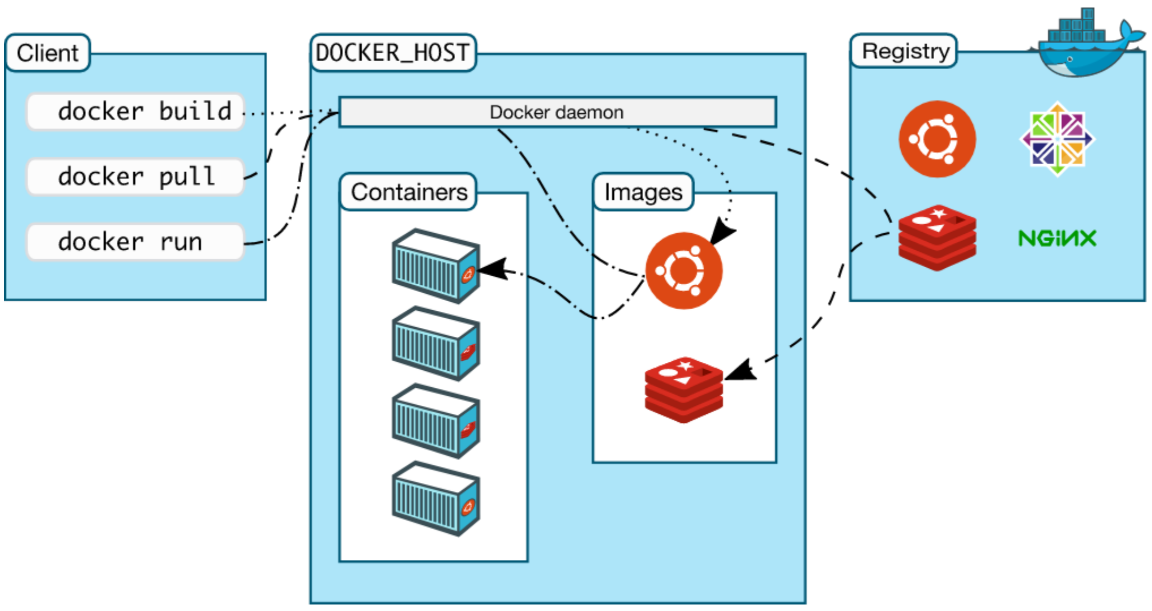


Figure 2. Docker architecture

Docker daemon

The Docker daemon, also known as *dockerd*, runs as a background process and listens for Docker API requests from the Docker client based on which the daemon will then manage the Docker objects to fit the requests (5).

Docker client

The Docker client, or just *docker*, is the main way users interact with Docker. The client reads user-inputted commands, usually from the command-line, and forwards them to the docker daemon to execute (6).

Docker objects

Docker objects are the entities that exists in the Docker system and are the ones manipuliated by the Docker daemon whenever it executes commands from users. This includes images, containers, networks, volumes, and more. The two most important objects to know are: the Docker image, and the Docker container (7).

- Docker images

A Docker image is a read-only blueprint with instructions for building a Docker container. As previously mentioned, an image is based on another base image, with some extra modifications. For instance, an image for building a container for a NodeJS application will usually start with the base image of NodeJS itself, and then add on instruction for building and running the actual application as well as its configuration.

- Docker containers

A Docker container is basically a runnable instance of a Docker image. Built from the instruction of a Docker image, the container is the application for which the image is the blueprint. A container can be freely controlled by the users through the Docker client or Docker API. Users can also manipulate a container's network or storage configurations.

Docker Registry

A Dockery *registry* is used to stores the physical copies of Docker images and is where Docker will look for images when creating new containers. The default and public registry for all Docker on installation is Docker Hub. User can also host their own private registries if they so choose. (8)

2.1.3. Docker as a container runtime

Eventhough often referred to as a container runtime, Docker is actually not a container runtime, but an entire tech stack. Docker uses a container runtime known as *containerd* below the surface, while provide high-level and user-friendly abstraction to manage containers on top of it. And in today's fast-growing tech world, alternatives to Docker can be found right around the corner. This is key reason for the shifting relationship between Docker and the popular container orchestration tool Kubernetes (9).

2.2. Kubernetes

Kubernetes, also known as K8s, is an open-source system for automating deployment, scaling, and management of containerized applications (10).

Originally developed at Google for their internal container orchestration system, K8s was then gifted by Google to the Cloud Native Computing Foundation (CNCF) and made open-sourced. Being one of the few graduated CNCF projects, K8s supports a plethora of features for orchestrating containerized applications. Some notable ones are:

- Horizontal scaling
- Automated rollouts and rollbacks
- Storage orchestration
- Service discovery and load balancing
- Secret and configuration management
- Self-healing

2.2.1. Container orchestration

What does container orchestration mean? In its simplest form, container orchestration is the automation of the deployments, scaling and networking of containerized applications. While containerization brings a lot of benefits onto enhancing application development and production, its effectiveness is limited by its use case. If containers were used for running monolithic applications, then it would only bring plain merits such as compatibility and portability. In reality, where containerization really shines, is when it is used in a microservice architecture. In this kind of architecture, an application is comprised of many smaller self-contained parts, or microservices. These can be containerized and run separately from each other, increasing the repairabilty and scalability as well as the availability of the application.

So, why container orchestration? The answer is as mentioned earlier, scalability. For production-grade and enterprise applications, which usually follows the microservice architecture, they do not require just one or a few containers, they require hundreds or thounsands. With no orchestration or management system in place, scaling such systems would be analogous to trying to fight a ten-front war with no general. Your troops, or in this case, the applications would not even make it out of their base. When Docker and containerization technology became increasingly popular and showed potential applications in today's extremely fast-growing world, Kubernetes step on to solve the scalability problem and provide a proper container orchestration platform, it in turn became one of the most successful and important open-sourced projects.

2.2.2. Relation with Docker

Back in its infancy, Kubernetes chose Docker to be its de-facto container runtime. This is due to a simple reason, there were none that could compete with Docker as a container runtime at that time. As containerization technology progressed, more and more Docker alternatives were brought into attention and showed prospective of performing better at being a container runtime than Docker. Combine with the previously mentioned fact, that Docker is in fact not a container runtime itself, Kubernetes officially announced its deprecation of Docker as a container runtime in late 2020 (9).

New versions of K8s (versions greater than 1.20) will no long support Docker as a container runtime but will be in favor of Container Runtime Interface (CRI) compliant container runtimes. One example of a CRI-compliant runtime is *containerd*, the same runtime that Docker originally used. This move by K8s is to remove Docker as the middleman between itself and the actual container runtime, and thus remove unnecessary complexity. However, this change does not actually affect the workflow of developers all that much.

Eventhough Docker is now no longer supported as a container runtime for Kubernetes, Docker images can still be used to build runnable containers on the CRI-compliant runtimes. This is because Docker images are OCI (Open Container Initiative) compliant, which is the main criteria for runnable containers of CRI-compliant runtimes. Hence, even after the deprecation of Docker by Kubenetes, Docker with its immense popularity still and will play an important role when developing applications for deploying to a K8s environment.

2.2.3. Kubernetes architecture

Kubernetes architecture may seem very complex at first glance, and it is complex, still it is nothing more than a lot of abstraction layers.

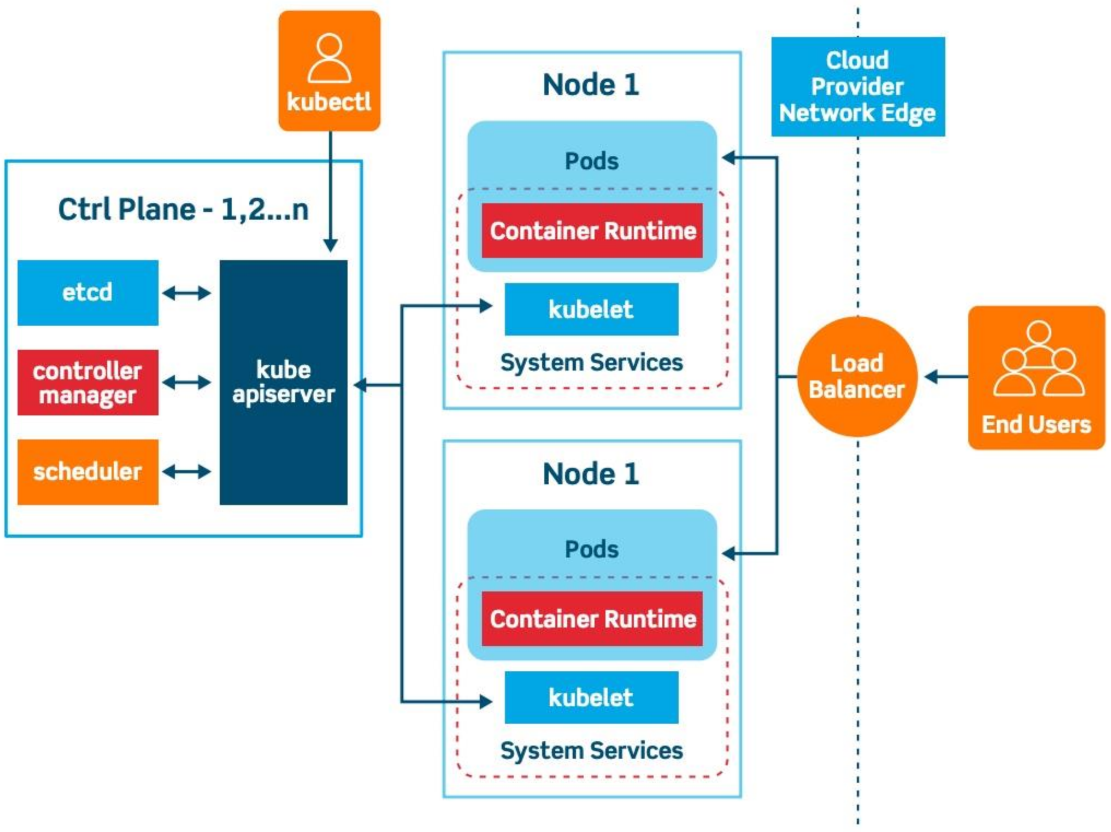


Figure 3. Kubernetes cluster architecture

Kubernetes cluster

Kubernetes architecture starts with something called a *cluster*. The cluster contains a set of *worker nodes* along with at least one *master node*.

Node

A node can be either a virtual or physical machine, and is managed by the control plane, or master nodes. Containers are put into pods and worker nodes are responsible for running those pods (11).

Control plane

Control plane, or master nodes, is responsible for managing and scheduling application containers across all clusters' nodes. These master nodes communicate with worker nodes through the K8s API server and the kubelet on each node (12).

Kubelet

Kubelets are processes running on worker nodes that are responsible for executing container management tasks as delegated to them by the control plane through the K8s API server. These tasks can be starting, stopping, checking or replacing application containers (13).

Pod

Pods are groups of one or more application containers which share storage and network resources. They are the smallest deployable computing unit that users can create and manage in a Kubernetes system (14).

Other components

Since the scale of the Kubernetes project is massive, it would be impossible to cover all of its architectural components in this report. However, some other Kubernetes resources that were important in the development of this thesis will be briefly discussed. These include:

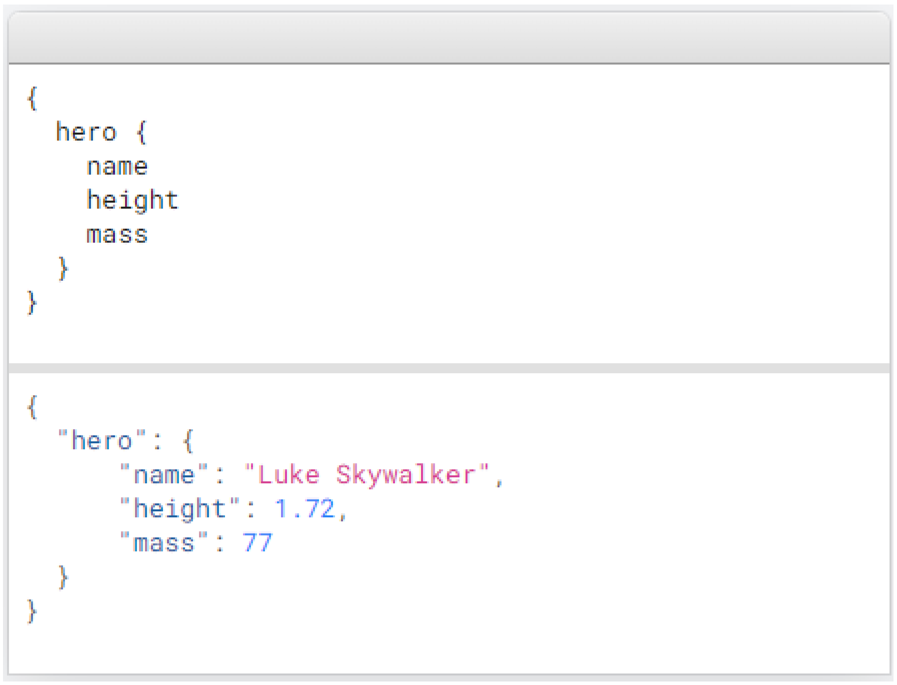
- **Deployments**: These are the most basic description for deploying an application in the Kubernetes cluster. It describes the desired state of the application being deployed as well as the pods which run the application containers (15).
- **StatefulSets**: The equivalent of deployments for stateful applications that requires synchronization of states across all instances such as databases (16).
- **ReplicaSet**s: ReplicaSets acts as the blueprint from which pods are created and is in charge of maintain and guarantee the availability of a stable set of replica pods (17).
- **Secrets**: Small objects that contains sensitive data such as service passwords, tokens or keys that are essential for application containers to connect to other services (18).
- **ConfigMaps**: The equivalent of secrets for non-confidential data (19).

2.3. GraphQL

GraphQL is a new standard for data query API between clients and servers in a client-server architecture system, with the old standard being the popular REST API. Unlike REST API, data queries in a GraphQL API are not predefined by the server providing the API, but rather the client which requests the data. The server only provides the schema of the API describing all possible data that can be queried to the client, and the client forms the data query using the schema provided. GraphQL also eliminates the need to extend the API with more specific entrypoints when the usage requires, which is very common in REST APIs. This brings many benefits to the table when working with a small monolithic application which needs to access many different types of data.

2.3.1. Redundant data reduction

Since clients can define their own queries, they can ask for exactly what needed to be display and query just that. This reduces the amount of redundant data in each request when compared to a REST API, the amount depends on how specific each query in the equivalent REST API is (20).



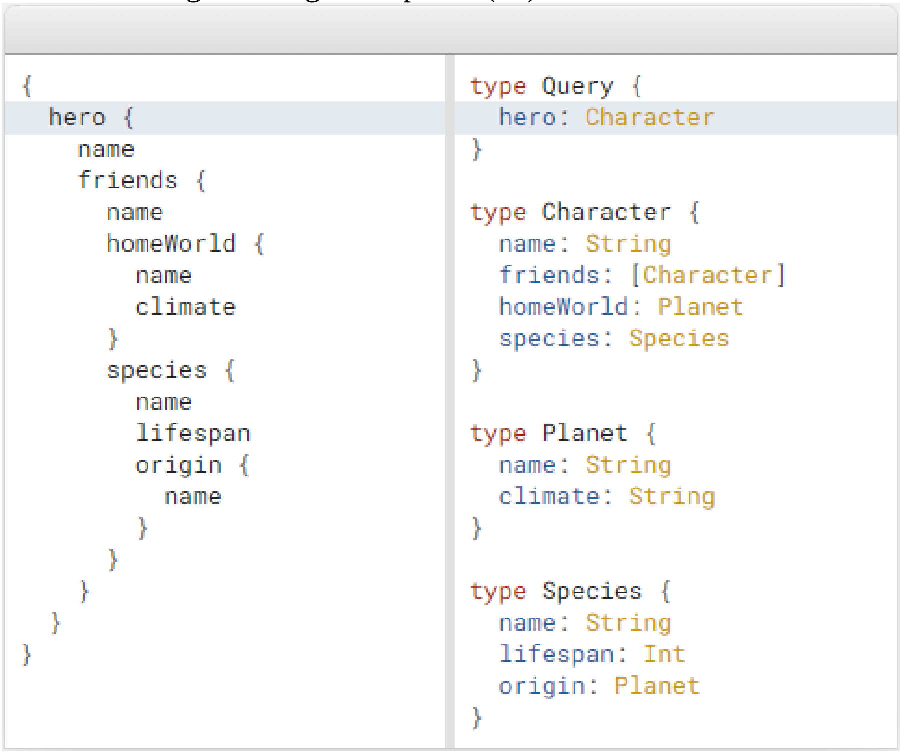
```
{
  hero {
    name
    height
    mass
  }
}
```

```
{
  "hero": {
      "name": "Luke Skywalker",
      "height": 1.72,
      "mass": 77
  }
}
```

Figure 4. GraphQL query example

2.3.2. Human-readable API

In GraphQL APIs, the schema for the data is provided beforehand, allowing higher readability and unity for developers during development. Backend developers only need to provide resolvers for the data defined in the schema while frontend developers only need to follow and create queries based on the predefined schema. This results in higher productivity and lower rate of error during the integration phase (20).



```
{
  hero {
    name
    friends {
      name
      homeWorld {
        name
        climate
      }
      species {
        name
        lifespan
        origin {
          name
        }
      }
    }
  }
}
```

```
type Query {
  hero: Character
}

type Character {
  name: String
  friends: [Character]
  homeWorld: Planet
  species: Species
}

type Planet {
  name: String
  climate: String
}

type Species {
  name: String
  lifespan: Int
  origin: Planet
}
```

Figure 5. GraphQL schema and query

2.3.3. GraphQL vs REST

Although GraphQL seems to have complete advantage over REST, it does not. In reality, depending on the purpose, each API standard has its own merits and drawbacks. For example, while GraphQL eliminates the need for supporting extra API endpoints when demand for more specific query arises, developing resolvers for GraphQL schemas can be very tedious and contains a lot of boilerplate code. On the other hand, REST API can also be very tedious to develop in queries are dynamic and needs to change rapidly depending on the state of the client. In their own words, GraphQL does not replace REST, but can co-exist in the same tech stack (21).

2.4. Related works

As previously mentioned in the introduction, there are no prevalent systems or solutions currently in use that can provide a suitable thesis software management. This is especially true in the context of domestic universities. Many universities, even established ones, have at best a simple list of theses as their theses management system, if at all. With that being said, there are systems and solution proposals that aim to solve parts of what this thesis aims to demonstrate. These similar works share the same direction as this thesis, either by being a thesis management system themselves, or by utilizing the same technologies for solving the software management problem.

2.4.1. Knative

Knative is an open-sourced community project for managing, deploying, and running serverless, cloud-native applications on Kubernetes (22).

Knative takes care of the server provision and management tasks, allowing developers to concentrate on their code without having to pay mind to the complex infrastructure setup. This boosts the software development productivity. Knative's cloud computing model also reduce the operational cost of running applications by being serverless, allowing the "pay as you go" model. With this model, users only pay based on their applications' compute time, not the total time they rent the hosting (22).

Eventhough Knative seems to be a very solid solution to the software management problem, with all its productive and financial benefits, Knative's main target is serverless, cloud-native applications. This is way too specific and limiting for a proper thesis management system, where software architecture is spontaneous due to the innovative nature of the education environment.

2.4.2. Other works

Unlike industry commercial-oriented projects such as Knative, projects and systems in the education field are much smaller in scale and technical complexity. As such, the capabilities of these systems are also limited, mostly stopping at managing basic information of theses and maybe thesis reports. Some examples are the thesis management system of universities such as the Keep Digital Repository Library used by VNU – HCMIU (23) or those used by foreign universities like MIT (24) or VUT (25). What all of these systems have in common is that they all serve as just basic digital libraries for old thesis documents and don't support the management of any software that comes with the theses.

Even so, the prospect of using containerization technology for software application on university's premise has been a point of interest for many learners. One such example is a bachelor's thesis of a student from Brno University of Technology (26). In this thesis, the process of setting up an on-premises mini cloud infrastructure using Kubernetes and Docker was explored and demonstrated the possibility of a ready-to-use cloud infrastructure for universities set up on their own grounds. Still, the thesis stops at only exploring the methodology of setting up an on-premises mini cloud infrastructure without delving into any practical applications. In our thesis, both the methodology and application of utilizing containerization technology to solve problems in the education field is studied. A draft thesis software management system is also designed and built to demonstrate the practical application of containerization technology.

CHAPTER 3

METHODOLOGY

Before the implementation of the thesis management system can be started, a suitable design needs to be drafted. In order to create a fitting design, a requirement analysis is necessary to understand the scope of what needs to be implemented in the system. A usecase diagram should be extracted from the requirements gathered from the analysis process to represent the functionalities needed in the system. With a use case diagram, the system and database structure can then be designed to support those functionalities. After the system design is drafted, development technologies are chosen, and assessment criteria is selected to grade the implementation.

3.1. Requirements Analysis

When designing a thesis management system capable of also managing the deployment and running of the thesis software, first the users (actors) of the system must be identified. Since the system is for managing theses, thesis owners are an obvious actor. The thesis management system needs to be able to allow thesis owners to manage their own theses software. This includes being able manage their own theses' information, being able to create and delete the resource definition for their theses' software.

This system should also be able to isolate the thesis software resource definitions of different theses in order to make sure that no thesis owner can tamper with another thesis that they don't own, regardless of whether it was deliberate or not. For this, software resource definitions will be separate per thesis using namespaces; each thesis will be their own namespace in the Kubernetes cluster. Thesis owners' access to namespace will be maintained in relational database, along with user information and thesis information. In order to manage the namespace, of which the management should be out of reach for normal thesis owners, another higher authorized actor is required. This actor is an admin, who can manage the creation, distribution, and destruction of namespaces. Admins should also be able to do what thesis owners can, which is to manage namespace's resource definitions and thesis information.

The last but not least important actor of the thesis management system is guests. Guests are those who will visit the platform to view, review thesis information or the demonstration of the theses' software. For this actor, the main 2 use cases are viewing thesis information, and viewing thesis demonstration.

3.1.1. Usecase description

A brief list of use cases for each actor can be extracted from the requirement analysis process:

- Admin:
    - Manage Namespaces
    - Manage Resources
    - Manage Users
- Thesis Owner:
    - Manage Owned Namespace
    - View Owned Namespace
    - Manage Thesis Info
- Guest:

- View Thesis Info
- View Thesis Demo

The description of each use case is summarized in Table 1.

Table 1. Usecase description table

Usecase	Description
Manage Namespaces	The actor should be able to create, delete and view namespaces in the Kubernetes cluster.
Manage Resources	The actor should be able to create, delete system resources in the Kubernetes cluster such as storage using Kubernetes volumes.
Manage Users	The actor should be able to create and delete user accounts, as well as assign to them namespaces.
Manage Owned Namespace	The actor should be able to create and delete software resource definitions in a Kubernetes namespace should they own that namespace.
View Owned Namespace	The actor should be to view all resource definitions made in a Kubernetes namespace should they own that namespace.
Manage Thesis Info	The actor should be able to create, edit and delete their thesis information.
View Thesis Info	The actor should be able to view the thesis information.
View Thesis Demo	The actor should be able to view the software demonstration of thesis.

3.1.2. Usecase diagram

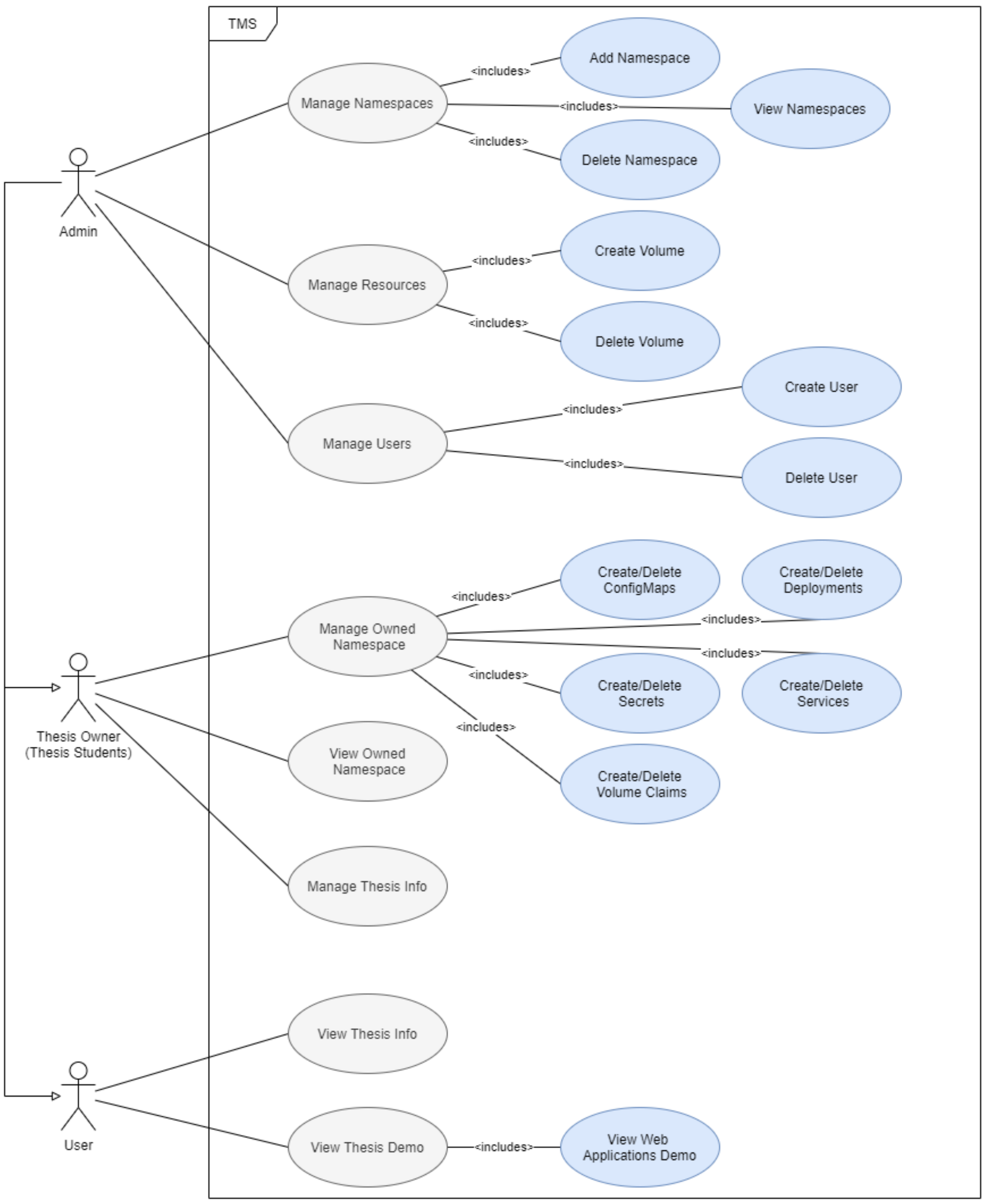


Figure 6. Usecase Diagram

3.2. System Design

As briefly discussed in earlier chapters, the software-based thesis management will be a web-based platform for its accessibility. The system design is splitted into 2 main components: A single page web application (SPA) serving as the front-end user interfact, and a backend API server to handle HTTP requests from the SPA. The backend server also automatically manages a Kubernetes Cluster according to requests from the web application. In addition, a relational database is employed to maintain basic user and thesis information, from which the API server will retrieve data and perform request logic processing.

3.2.1. System architecture

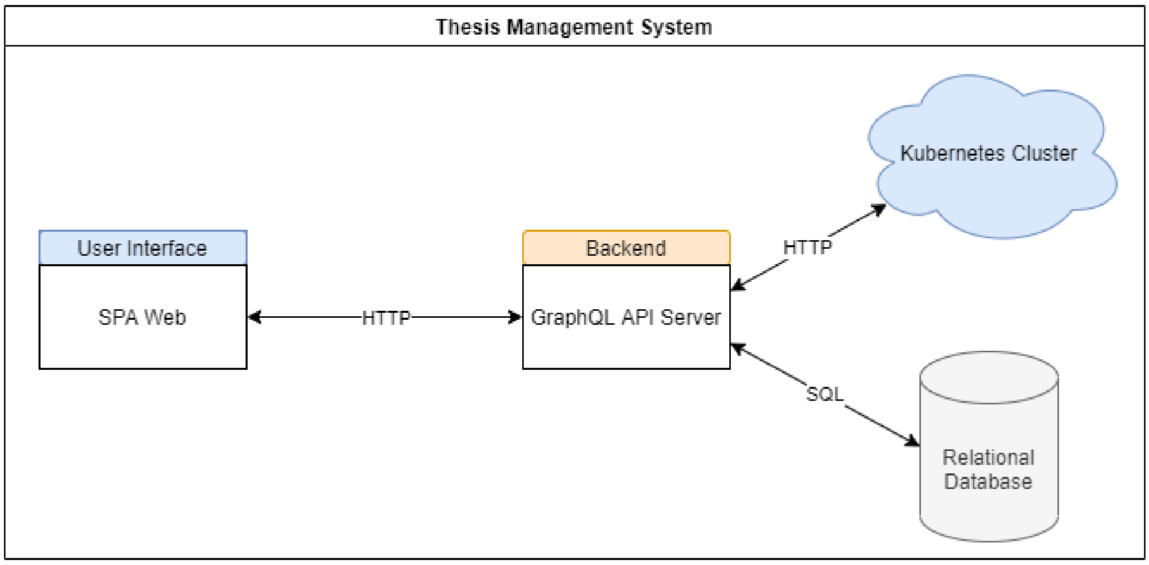


Figure 7. System architecture

As previously explained, the system will have 4 components in total:

- a Front-end Single Page Web Application: to handle user interactions
- a Back-end API Server: to handle front-end requests and manage Kubernetes as well as thesis data.
- a Kubernetes Cluster: to run and orchestrate theses' software containers.
- a Relational Database: to store user and thesis data.

To handle user interactions, a Single Page Application (SPA) is used as the frontend client. This web client will provide a user-friendly interface for guest users to view thesis information and demos, for thesis owners to manage their thesis information and software resources, and for administrator to manage users, namespaces and hardware resources. The use of SPA will allow users to have a much smoother experience and will increase the user experience.

In order to handle the requests sent by the frontend SPA client, a backend GraphQL API server is used. The GraphQL API will act as intermediary between the frontend and backend, allowing the communication of requests and data. The backend server also communicates with the Kubernetes cluster through a Kubernetes client to perform management on the resources when needed.

For running thesis software containers, a Kubernetes cluster is used. The k8s cluster is responsible for automatically deploy and run the containerized thesis software applications when demanded by the backend server. By design, the Kubernetes cluster will be demanded to scale down all software by the backend server when it is idle to save resources. In addition, each thesis will be given its own namespace in the Kubernetes cluster, allowing environment isolation between theses and stopping all interference.

To store the thesis and user data, a relational database is used. There are many different archetypes of database on the market, but a relational database is chosen due to its simplicity as well as to avoid unnecessary time wasted on learning another database.

3.2.2. Database diagram

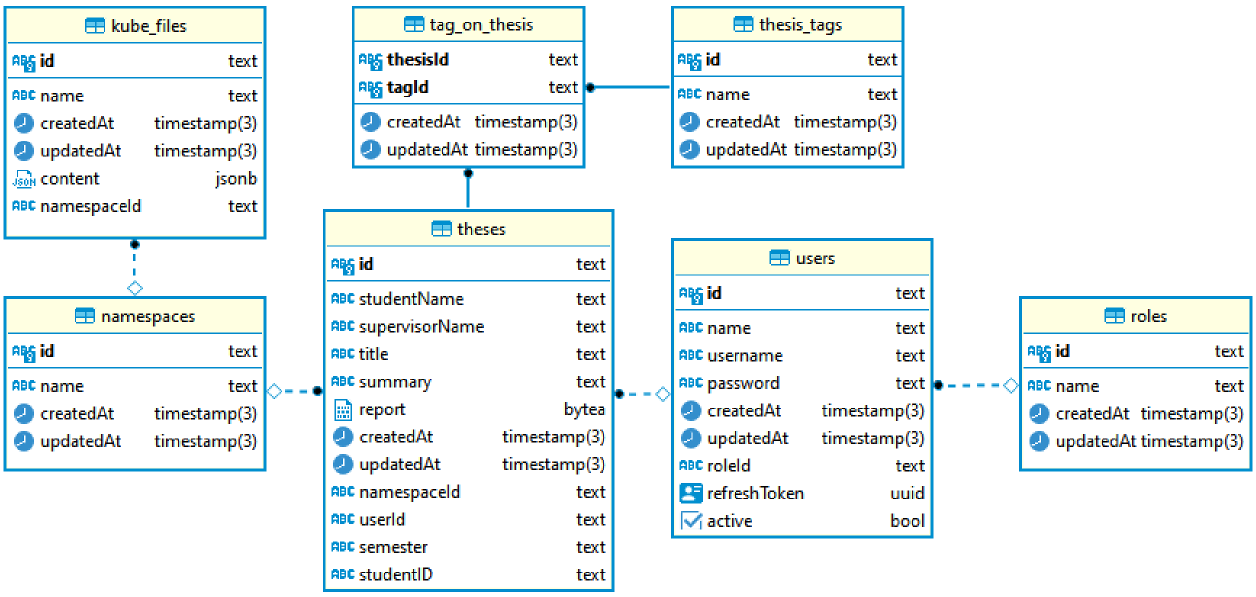


Figure 8. Entity Relationship Diagram

The design for the database is shown in Figure 8. The fields and relations are carefully chosen to make sure that every table satisfy at least the Boyce-Codd normal form (BCNF). The proof is not included in this report since the database design is not the main point of concern, however further verifications are always welcomed.

It is important to notice that each thesis is given a user and a namespace. This is an integral part of the system design; each namespace will correlate with an actual namespace of the Kubernetes cluster and will provide a sandbox environment for each user's thesis software. Every namespace also has relations with a number of KubeFiles, these are the Kubernetes resource definitions for all of the resources in the namespace and will be persisted on creation of those resources.

3.3. Kubernetes Cluster Strategy

A Kubernetes cluster is needed to run and orchestrate all thesis software containers. For this, Minikube will be used for the sake of simplicity in setting up the Kubernetes cluster. In addition, the Minikube registry addon will be used to store container images created from the source code of thesis projects using Docker.

3.3.1. Minikube

Minikube is a simple, local Kubernetes cluster focused on making Kubernetes easy to develop and test on the local machine. For this thesis, minikube is sufficient as a Kubernetes cluster. However, other Kubernetes clusters will still work the same with the thesis management system proposed.

3.3.2. Registry Addon

In order for Kubernetes to run the containers, the images of which to build are needed to be maintained. There are a few options for maintaining the images of containers such as using Docker Hub public registry or self-hosting a private registry. Alternatively, Minikube's registry addon can be used. This addon runs a registry container right inside the Kubernetes cluster, acting as a private registry and hosting the image of all containers running inside the Kubernetes cluster.

3.4. Backend Strategy

In order to implement the backend API server, TypeScript is chosen as the main developing language. This is due to TypeScript being a superset of JavaScript that enforces type-safety features, allowing for both less error-prone development and JavaScript's extensive web framework support. NodeJs has also become quite popular in the industry's server space due to it being a JavaScript runtime, seeing major usage in modern web application tech stack. Therefore, NodeJs is chosen to run the API server.

Due to the benefits of GraphQL API discussed in section 2.3, GraphQL is chosen to be the API standard for communicating between the frontend and backend. This meant the API server must be able to serve GraphQL API requests along with HTTP requests. For this, Apollo GraphQL is chosen since it's the most popular framework and one of the ones recommended by the official GraphQL documentation.

3.4.1. NodeJS

NodeJs is an asynchronous event-driven JavaScript runtime, designed to build scalable network applications. It can handle multiple concurrent connections efficiently thanks to the event-driven architecture rather than the thread-based concurrency model used by others, reducing computation power wasted when there are no connections. (27) The NodeJs version used in this system is v16.

3.4.2. Apollo GraphQL Server

Apollo GraphQL Server is one of the most popular GraphQL server implementation, providing the platform for build GraphQL API servers. In this system, Apollo Server is used in junction with Express, the de facto backend web application framework for Javascript. This allows the server to handle both GraphQL API requests and normal HTTP requests.

3.4.3. Prisma

Prisma is an open-source database toolkit for NodeJs, providing type-safe ORM capability for TypeScript & NodeJs. It is also incredibly helpful in migrating and change database schema, as well as generating custom database connector for TypeScript.

3.4.4. Kubernetes TypeScript Client

To manage the Kubernetes cluster on which the system lies, the official Kubernetes TypeScript Client is utilized. This client provides a generous number of features for controlling resources on the Kubernetes cluster through HTTPs communication with the control plane. In other words, the Kubernetes TypeScript Client provides an API similar to the kubectl tool that comes with every Kubernetes installation. (28)

3.5. Frontend Strategy

The frontend application will be, as mentioned above, a Single Page Application (SPA). For this, the designated development language is also TypeScript. This is an easy decision since most modern UI frameworks are native to JavaScript, which is a subset of TypeScript. As for which UI framework to use, ReactJS is chosen. This is due to React's popularity resulting in its wide community support, which makes the development process more efficient.

3.5.1. Single Page App

Single Page Applications are one of the latest developments in the field of web application. Unlike traditional web applications which reload and re-render the entire HTML page everytime the user performs an action like submitting a form or search for a post, SPA does not need to reload the entire page. SPAs achieve this through sending AJAX requests for the data and re-rendering only the affected view models using JavaScript straight on the user's browser.

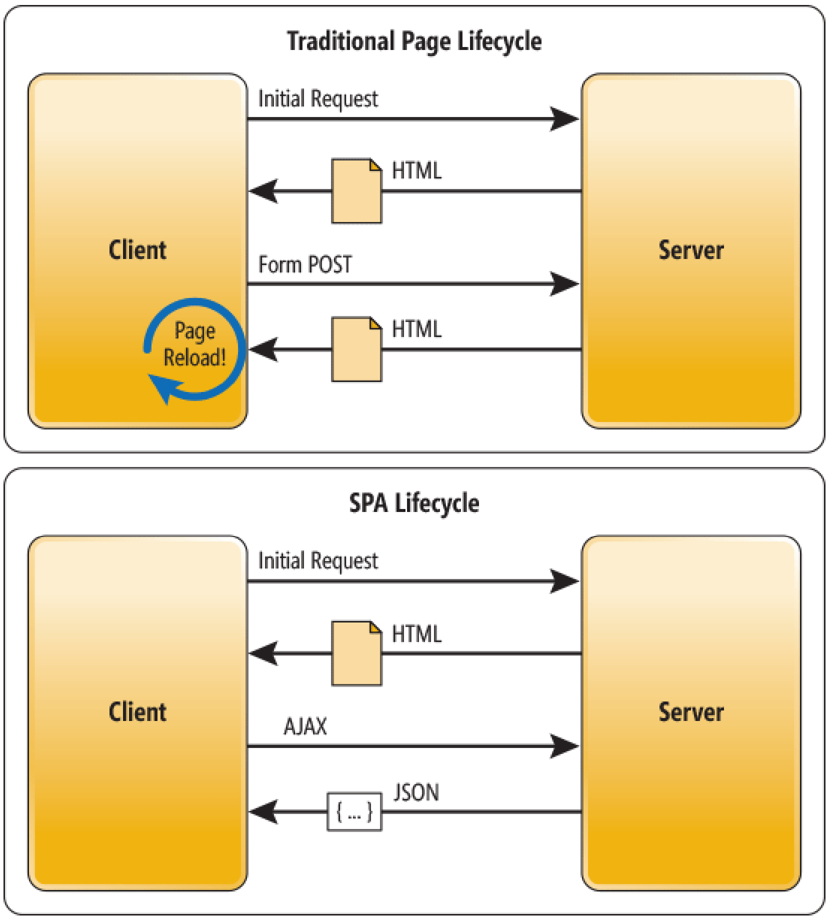


Figure 9. SPA vs Tradition web Lifecycle

Figure 9 shows the lifecycles of SPAs and traditional webpages. Due to the more efficient re-render, SPA usually has some distinguished benefits over traditional webpages:

- Fast and responsive
- Better local caching
- Smoother user experience

3.5.2. ReactJS

React is a popular free and open-source front-end JavaScript/TypeScript framework, famous for creating some of the most visited websites on the internet such as Facebook, Instagram, or Pinterest. It has a wide community support and is maintained by its founder Meta (formerly known as Facebook), making React easily accessible, and easy to learn and develop with. React also has extensive capabilities, able to be used as a base in a development of many types of applications such as Single Page Applications or mobile applications. As such, React is chosen to be the main UI framework for the web-based thesis management system.

3.6. Assessment criteria

Since the proposed thesis management system has no similar counterparts, as described in Chapter 1 and Chapter 2. This constitutes towards the assessment criteria having no baseline comparison and thus must rely solely on common statistics and completeness of the system. As such, to derive the performance assessment criteria for the system, statistics from an analysis released by Backlinko (29) were used. According to the statistics, on average a web page takes around 8 seconds to be visually complete and 10 seconds to be fully loaded. It is clear that just beating the average statistics would bring little meaning to the assessment, so the criteria will be defined as follow:

- Performance:
  - Web pages take less than 4 seconds to fully load.
  - Software deployment takes a reasonable amount of time, preferably less than 1 minute (Reasonable was used because the deployment time depends on what software was deployed).
- Functional:
  - System can support for all use cases defined in section 3.1.

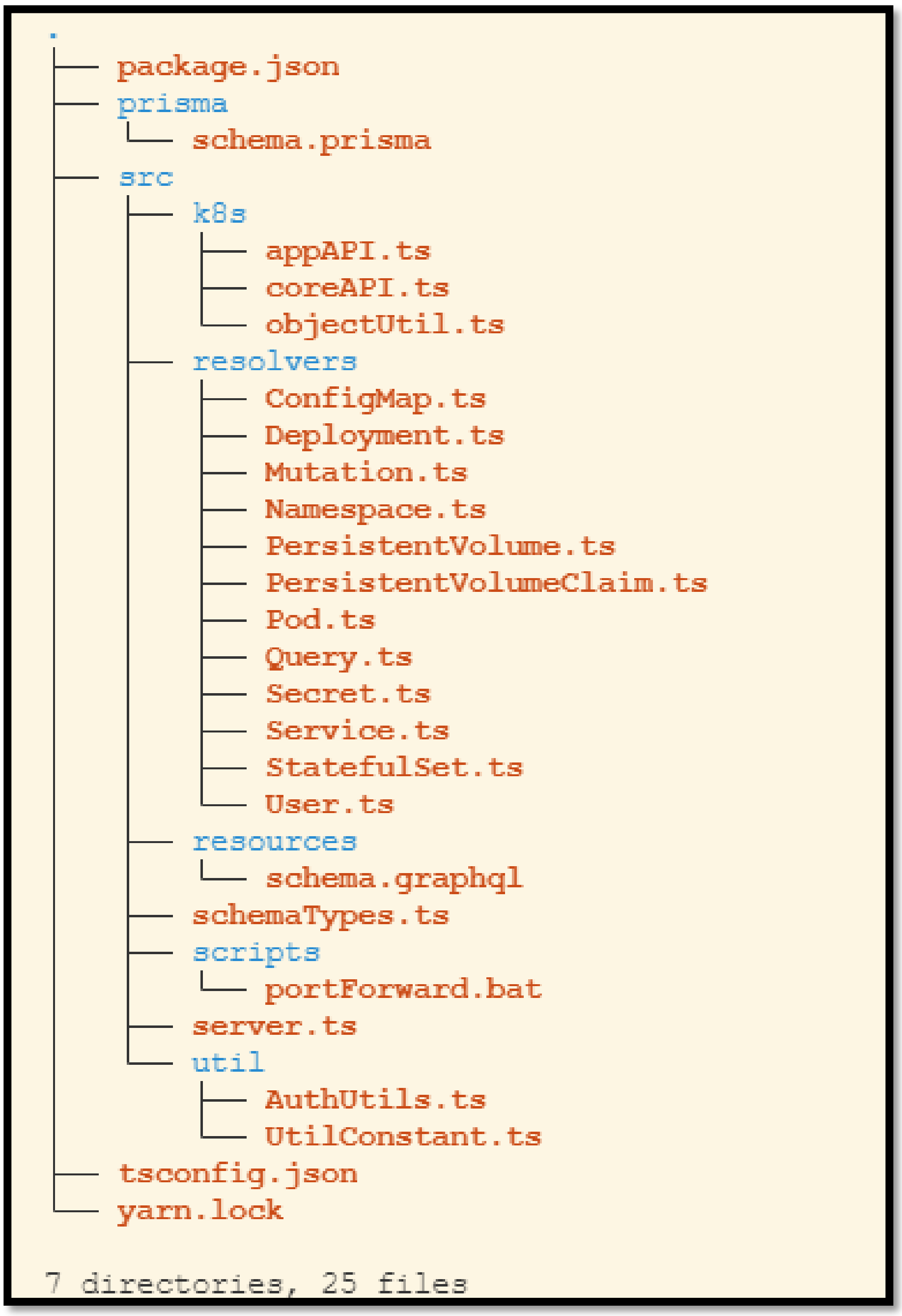
CHAPTER 4

IMPLEMENTATION

The implementation of the thesis management system is divided into 4 main parts:

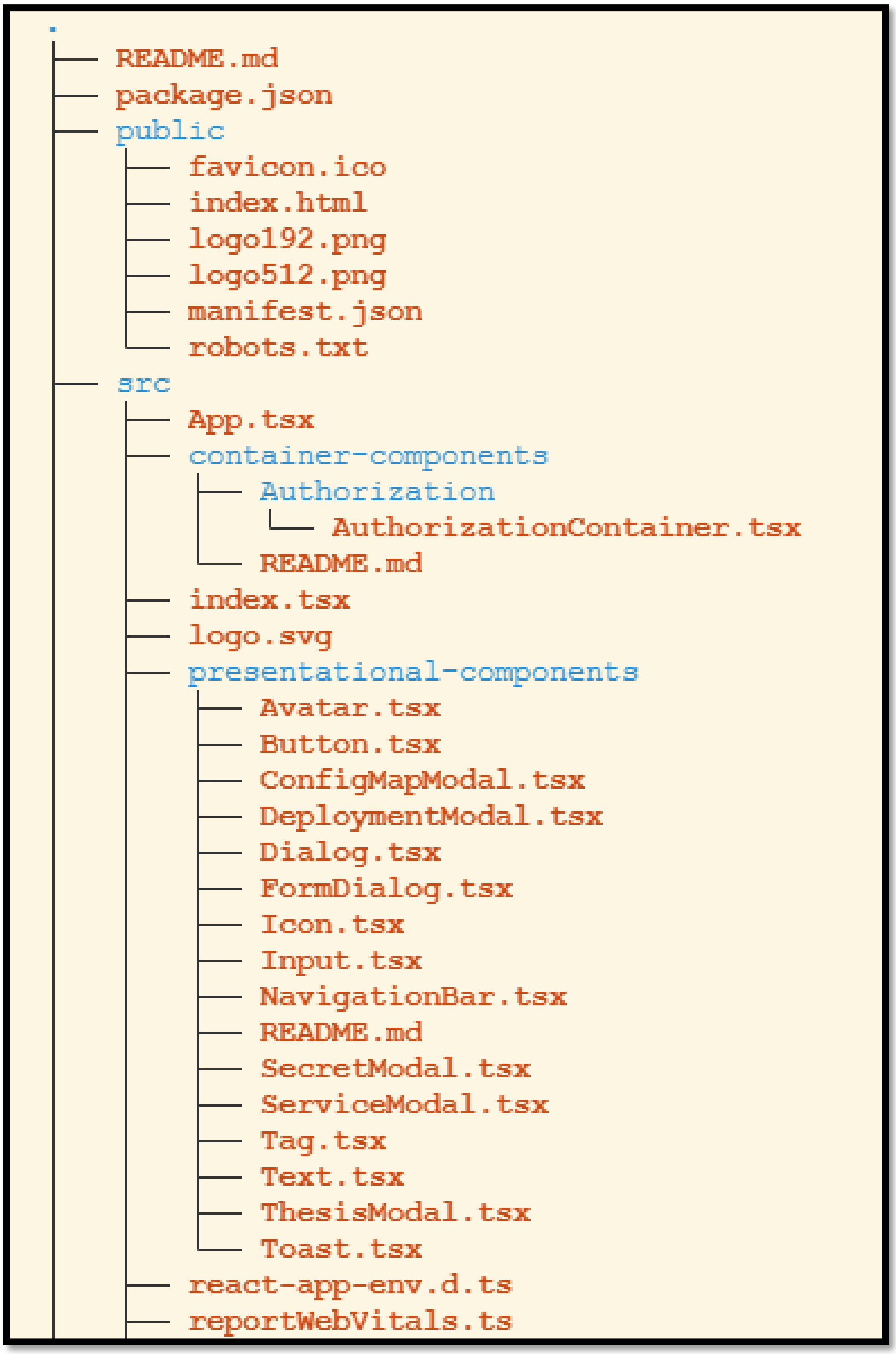
- The Kubernetes Cluster
- The Relational Database
- The Backend API Server
- The Frontend SPA Web App

The first 2 parts require only setup and running commands and does not need any coding. The latter 2 parts is coded from scratch and the structure for those 2 modules is described by the figures below.



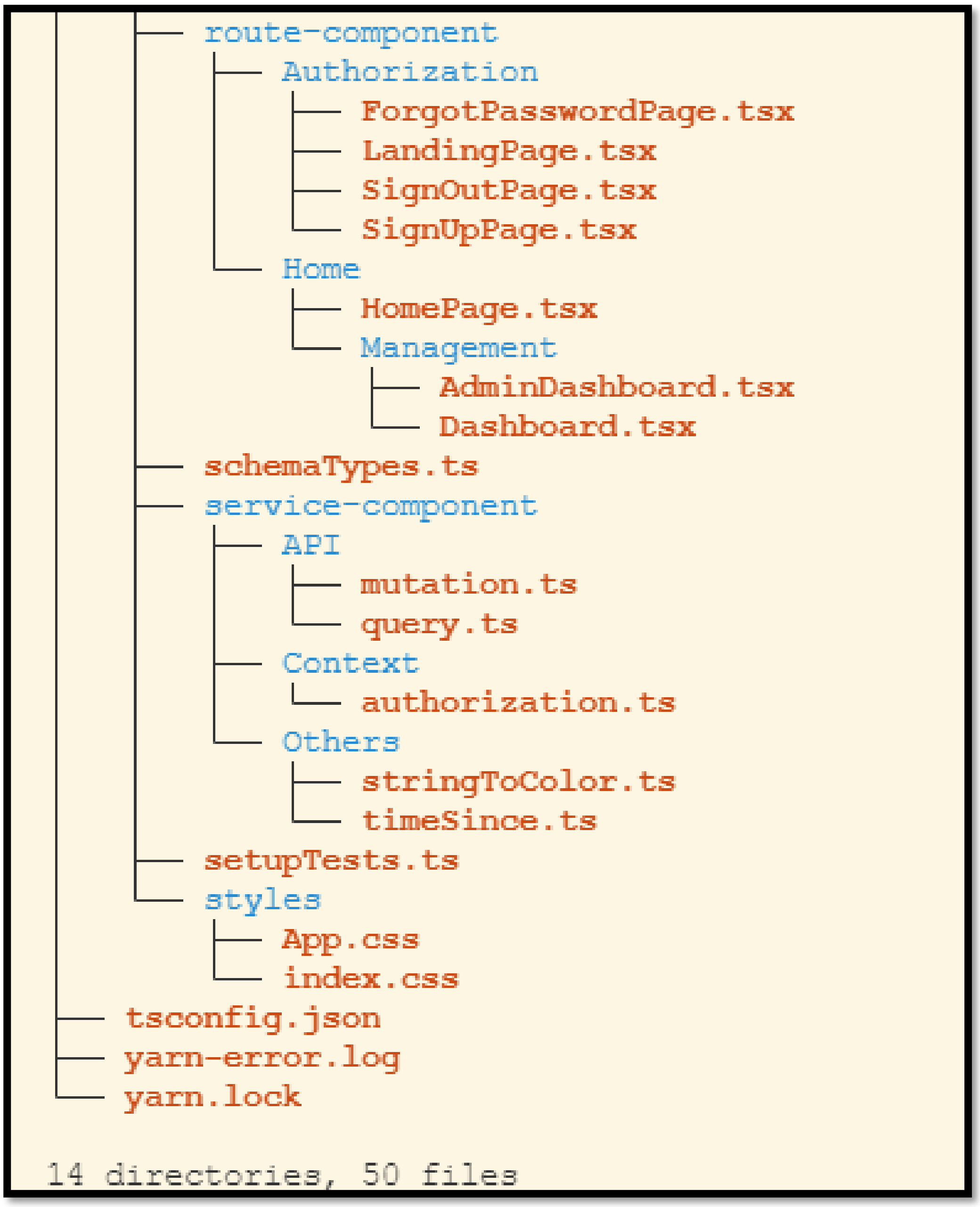
```
.
├── package.json
├── prisma
│   └── schema.prisma
├── src
│   ├── k8s
│   │   ├── appAPI.ts
│   │   ├── coreAPI.ts
│   │   └── objectUtil.ts
│   ├── resolvers
│   │   ├── ConfigMap.ts
│   │   ├── Deployment.ts
│   │   ├── Mutation.ts
│   │   ├── Namespace.ts
│   │   ├── PersistentVolume.ts
│   │   ├── PersistentVolumeClaim.ts
│   │   ├── Pod.ts
│   │   ├── Query.ts
│   │   ├── Secret.ts
│   │   ├── Service.ts
│   │   ├── StatefulSet.ts
│   │   └── User.ts
│   ├── resources
│   │   └── schema.graphql
│   ├── schemaTypes.ts
│   ├── scripts
│   │   └── portForward.bat
│   ├── server.ts
│   └── util
│       ├── AuthUtils.ts
│       └── UtilConstant.ts
├── tsconfig.json
└── yarn.lock

7 directories, 25 files
```

Figure 10. Backend code structure



```
.
├── README.md
├── package.json
├── public
│   ├── favicon.ico
│   ├── index.html
│   ├── logo192.png
│   ├── logo512.png
│   ├── manifest.json
│   └── robots.txt
├── src
│   ├── App.tsx
│   ├── container-components
│   │   ├── Authorization
│   │   │   └── AuthorizationContainer.tsx
│   │   └── README.md
│   ├── index.tsx
│   ├── logo.svg
│   ├── presentational-components
│   │   ├── Avatar.tsx
│   │   ├── Button.tsx
│   │   ├── ConfigMapModal.tsx
│   │   ├── DeploymentModal.tsx
│   │   ├── Dialog.tsx
│   │   ├── FormDialog.tsx
│   │   ├── Icon.tsx
│   │   ├── Input.tsx
│   │   ├── NavigationBar.tsx
│   │   ├── README.md
│   │   ├── SecretModal.tsx
│   │   ├── ServiceModal.tsx
│   │   ├── Tag.tsx
│   │   ├── Text.tsx
│   │   ├── ThesisModal.tsx
│   │   └── Toast.tsx
│   ├── react-app-env.d.ts
│   ├── reportWebVitals.ts
```

Figure 11. Frontend code structure (Pt. 1)



```
│   ├── route-component
│   │   ├── Authorization
│   │   │   ├── ForgotPasswordPage.tsx
│   │   │   ├── LandingPage.tsx
│   │   │   ├── SignOutPage.tsx
│   │   │   └── SignUpPage.tsx
│   │   └── Home
│   │       ├── HomePage.tsx
│   │       └── Management
│   │           ├── AdminDashboard.tsx
│   │           └── Dashboard.tsx
│   ├── schemaTypes.ts
│   ├── service-component
│   │   ├── API
│   │   │   ├── mutation.ts
│   │   │   └── query.ts
│   │   ├── Context
│   │   │   └── authorization.ts
│   │   └── Others
│   │       ├── stringToColor.ts
│   │       └── timeSince.ts
│   ├── setupTests.ts
│   └── styles
│       ├── App.css
│       └── index.css
├── tsconfig.json
├── yarn-error.log
└── yarn.lock

14 directories, 50 files
```

Figure 12. Frontend code structure (Pt. 2)

The implementation source code can also be found at https://github.com/Nitaray/thesis-management-system

4.1. Kubernetes Cluster

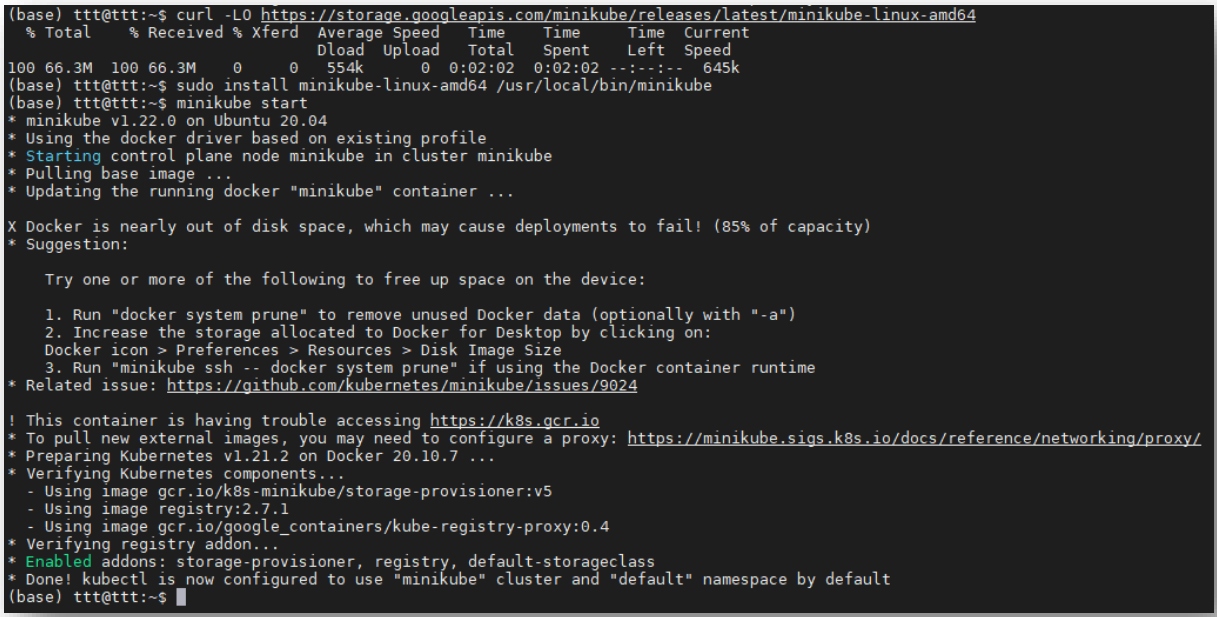
First, the Kubernetes Cluster is installed using Minikube as mentioned in section 3.3. Following the guide on minikube's official website, installing is quite trivial.

```
curl -LO https://storage.googleapis.com/minikube/releases/latest/minikube-linux-amd64
sudo install minikube-linux-amd64 /usr/local/bin/minikube
```

After minikube finished installing, the only step left is to start it with

```
minikube start
```

The Kubernetes cluster and kubectl will be configured and ready to use once the starting process is done.



```
(base) ttt@ttt:~$ curl -LO https://storage.googleapis.com/minikube/releases/latest/minikube-linux-amd64
  % Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
                                 Dload  Upload   Total   Spent    Left  Speed
100 66.3M  100 66.3M    0     0   554k      0  0:02:02  0:02:02 --:--:--  645k
(base) ttt@ttt:~$ sudo install minikube-linux-amd64 /usr/local/bin/minikube
(base) ttt@ttt:~$ minikube start
* minikube v1.22.0 on Ubuntu 20.04
* Using the docker driver based on existing profile
* Starting control plane node minikube in cluster minikube
* Pulling base image ...
* Updating the running docker "minikube" container ...

X Docker is nearly out of disk space, which may cause deployments to fail! (85% of capacity)
* Suggestion:

    Try one or more of the following to free up space on the device:

    1. Run "docker system prune" to remove unused Docker data (optionally with "-a")
    2. Increase the storage allocated to Docker for Desktop by clicking on:
    Docker icon > Preferences > Resources > Disk Image Size
    3. Run "minikube ssh -- docker system prune" if using the Docker container runtime
* Related issue: https://github.com/kubernetes/minikube/issues/9024

! This container is having trouble accessing https://k8s.gcr.io
* To pull new external images, you may need to configure a proxy: https://minikube.sigs.k8s.io/docs/reference/networking/proxy/
* Preparing Kubernetes v1.21.2 on Docker 20.10.7 ...
* Verifying Kubernetes components...
  - Using image gcr.io/k8s-minikube/storage-provisioner:v5
  - Using image registry:2.7.1
  - Using image gcr.io/google_containers/kube-registry-proxy:0.4
* Verifying registry addon...
* Enabled addons: storage-provisioner, registry, default-storageclass
* Done! kubectl is now configured to use "minikube" cluster and "default" namespace by default
(base) ttt@ttt:~$ 
```

Figure 13. Installing and starting minikube

4.2. Database & ORM

To implement the database design in the previous chapter, a relational database is required. Being one of the classic types of databases, there are many existing database software satisfy this requirement, such as MySQL, SQLite, Microsoft SQL Server, Oracle Database, etc. For this implementation however, PostgreSQL will be used for its wider functionality pool as well as scalability. Even though this experiment is implemented on a local scale only, the implication of this system's approach to solving the software management problem is on a much wider scale. This means that this concept should also work on a larger scale system, and thus scalability is considered.

Since Kubernetes is an integral part of the system, the PostgreSQL database can be hosted directly on the local Kubernetes cluster. For this, the PostgreSQL official Docker container from Docker Hub is used.

First, a storage volume and claim need to be created on the Kubernetes cluster's default namespace. This storage volume will be used to persist the data of the database, allowing the database to retain its data in case failure occurs. As for why the default namespace was used, as mentioned in section 3.2.1, each thesis will be given a unique namespace in the namespace in the Kubernetes cluster. This allows for complete segregation between thesis software and isolation between different theses. The volume and volume claim are created using a Kubernetes Resource Definition YAML file.

```
apiVersion: v1
kind: PersistentVolume
metadata:
  name: local-volume
  namespace: default
spec:
  capacity:
    storage: 2Gi
  accessModes:
    - ReadWriteOnce
  persistentVolumeReclaimPolicy: Retain
  storageClassName: manual
  hostPath:
    path: "/mnt/data"
---

apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: local-volume-claim
  namespace: default
spec:
  accessModes:
  - ReadWriteOnce
  storageClassName: manual
  volumeMode: Filesystem
  resources:
    requests:
      storage: 1Gi
```

volume.yml

Even though Kubernetes supports a lot of different storage solutions, the local host filesystem is used in this implementation for simplicity purposes. The YAML file can be applied to the cluster using:

```
kubectl apply -f volume.yml
```

Then, a PostgreSQL deployment and service can be creating using the official PostgreSQL Docker image and the created volume. This is done by applying the YAML files:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: postgresql-deployment
  labels:
    app: postgresql
```

```yaml
spec:
  replicas: 1
  revisionHistoryLimit: 0
  selector:
    matchLabels:
      app: postgresql
  template:
    metadata:
      labels:
        app: postgresql
    spec:
      containers:
      - name: postgresql
        image: postgres
        resources:
          limits:
            memory: "128Mi"
            cpu: "500m"
        volumeMounts:
          - name: postgresql-data
            mountPath: /var/lib/postgresql/data
        ports:
        - containerPort: 5432
        env:
        - name: POSTGRES_USER
          valueFrom:
            secretKeyRef:
              name: postgresql-secret
              key: postgres-username
        - name: POSTGRES_PASSWORD
          valueFrom:
            secretKeyRef:
              name: postgresql-secret
              key: postgres-password
      volumes:
      - name: postgresql-data
        persistentVolumeClaim:
          claimName: local-volume-claim

---
apiVersion: v1
kind: Service
metadata:
  name: postgresql-service
spec:
  selector:
    app: postgresql
  ports:
    - protocol: TCP
      port: 5432
      targetPort: 5432
```

postgres.yaml

```
apiVersion: v1
kind: Secret
metadata:
  name: postgresql-secret
type: Opaque
data:
  postgres-username: dXNlcg==
  postgres-password: dXNlcg==
  postgres-port: NTQzMg==
```

postgres-secret.yaml

The latter file is used to define the secret environment variables for the PostgreSQL container and needed to be applied first. After this is done, the database is deployed on the Kubernetes cluster's default namespace and is ready to be used. The final step needed is to create a schema for this database, this is done using Prisma with the designed schema:

```
generator client {
  provider = "prisma-client-js"
}

datasource db {
  provider = "postgresql"
  url      = env("DATASOURCE_URL")
}

model Answer {
  AnswerId         Int                  @id @default(autoincrement())
  AnswerText       String
  AnswerHistory    AnswerHistory[]
  AnswerOfQuestion AnswerOfQuestion[]
}

model AnswerHistory {
  UserId       Int
  QuestionId   Int
  UserAnswerId Int
  Question     Question @relation(fields: [QuestionId], references:
[QuestionId])
  Answer       Answer   @relation(fields: [UserAnswerId], references:
[AnswerId])
  User         User     @relation(fields: [UserId], references: [UserId])

  @@id([UserId, QuestionId])
}

model AnswerOfQuestion {
  AnswerId   Int
  QuestionId Int
  IsCorrect  Boolean
  Answer     Answer   @relation(fields: [AnswerId], references:
[AnswerId])
  Question   Question @relation(fields: [QuestionId], references:
[QuestionId])

  @@id([AnswerId, QuestionId])
}

model Comment {
  CommentId       Int      @id @default(autoincrement())
  Content         String
  DateCreated     DateTime @default(now()) @db.Date
  CommentedUserId Int
  InTestId        Int
  User            User     @relation(fields: [CommentedUserId],
references: [UserId])
  Test            Test     @relation(fields: [InTestId], references:
[TestId])
}

model HasDone {
  UserId      Int
  TestId      Int
  StartTime   DateTime @default(now()) @db.Timestamp(6)
  ScoreResult Float?   @default(0)
  Test        Test     @relation(fields: [TestId], references: [TestId])
  User        User     @relation(fields: [UserId], references: [UserId])

  @@id([UserId, TestId])
}

model Question {
  QuestionId       Int                  @id @default(autoincrement())
  QuestionType     String
  Statement        String
  AnswerHistory    AnswerHistory[]
  AnswerOfQuestion AnswerOfQuestion[]
  QuestionInGroup  QuestionInGroup[]
}

model QuestionGroup {
  QuestionGroupId Int                 @id @default(autoincrement())
  IntroText       String
  TestSectionId   Int
  TestSection     TestSection         @relation(fields: [TestSectionId],
references: [TestSectionId])
  QuestionInGroup QuestionInGroup[]
}

model QuestionInGroup {
  QuestionGroupId   Int
  QuestionId        Int
  QuestionNumbering Int
  QuestionGroup     QuestionGroup @relation(fields: [QuestionGroupId],
references: [QuestionGroupId])
  Question          Question      @relation(fields: [QuestionId],
references: [QuestionId])

  @@id([QuestionGroupId, QuestionId])
}

model Role {
  RoleId   Int    @id @default(autoincrement())
  RoleName String
  User     User[]
}

model Test {
  TestId      Int           @id @default(autoincrement())
  TestType    String
  Title       String        @unique
  Comment     Comment[]
  HasDone     HasDone[]
  TestSection TestSection[]
}

model TestSection {
  TestSectionId   Int             @id @default(autoincrement())
  TestSectionType String
  StatementText   String?
  TestId          Int
  StatementAudio  String?
  Test            Test            @relation(fields: [TestId], references:
[TestId])
  QuestionGroup   QuestionGroup[]
}

model User {
  UserId        Int             @id @default(autoincrement())
  Username      String          @unique
  Fullname      String
  Password      String
  Rating        Int
  RoleId        Int
  RefreshToken  String?         @unique @db.Uuid
  Role          Role            @relation(fields: [RoleId], references:
[RoleId])
  AnswerHistory AnswerHistory[]
  Comment       Comment[]
  HasDone       HasDone[]
}
```

schema.prisma

Using the schema defined in a schema.prisma file, Prisma can automatically create the corresponding schema on the actual database using the command:

```
npx prisma db push
```

Prisma will automatically generate the schema in the database as well as the ORM client corresponding with this schema in the code. This greatly reduces the time taken to create and manage a database connector from scratch. With this, the database is ready to be used by the system for all its intended purposes.

4.3. Backend API Server

For the backend server, since it's a NodeJs server, the first thing to do is to install NodeJs.

```
curl -fsSL https://deb.nodesource.com/setup_lts.x | sudo -E bash -
sudo apt-get install -y nodejs
```

After installing NodeJs, the dependencies needed for the package can be installed. This is done with Yarn, a package manager for NodeJs, similar to the built-in npm. Unlike npm, yarn offers better performance and package caching. Yarn is first installed with npm.

```
npm install --global yarn
```

With Yarn as the package manager, dependencies for the server can be installed with

```
yarn add <dependency>
```

The full list of dependencies used can be found at the *Backend package.json* file in the Appendix. After the dependencies has been installed, the server's development is split into 2 main parts:

- Kubernetes API client
- GraphQL API server

4.3.1. Kubernetes API Client

Using the official client API provided by Kubernetes, a fitting Kubernetes client for the management system is developed. This client's source is located at the *k8s* directory shown in Figure 10 and has 2 main modules:

- CoreAPI
- AppAPI

These 2 modules correspond to the actual Kubernetes API versions used by the resources within the cluster. Each module is responsible for handling the CRUD actions of the resources of that API version in the Kubernetes cluster.

The CoreAPI contains the following resources:

- Namespace

- Service
- Pod
- Secret
- Config Map
- Persistent Volume
- Persistem Volume Claim

The AppAPI contains the following resources:

- Deployment
- StatefulSet

The aim of this k8s client module is to drastically reduces the complexity of the Kubernetes resource definition API to a more specific API scheme suitable for the thesis management system. This client omits a vast majority of resources in the Kubernetes cluster as well as provide better skeletons for CRUD operations of Kubernetes resources. In short, the developed client serves to make creating resources in the Kubernetes cluster less complicated by hardwiring system-specific values into the API calls and wrap them in another API.

4.3.2. GraphQL API Server

The GraphQL API Server is developed using the Apollo GraphQL Express Server, this server is a webserver responsible for processing HTTP requests from the frontend client and returning the correct data as well as perform any Kubernetes CRUD operation using the Kubernetes API client if necessary. The API schema for this GraphQL Server can be found at the schema.graphql file in the Appendix.

The server configurations are defined in the server.ts in the src directory, while all of the resolvers, or the request logic processing modules, are defined in the src/resolvers directory as can be seen in Figure 10. Other utilities modules are also developed to help with the development process.

In total, 12 resolver modules were developed to handle the processing of APIs defined in the schema, including:

- Query
- Mutation
- User
- Namespace
- Service
- Secret
- ConfigMap
- Deployment
- StatefulSet
- PersistentVolume
- PersistentVolumeClaim
- Pod

For the APIs defined in the schema, these 12 resolvers are sufficient to handle all requests and are tested. However, it is fair to acknowledge that more resolvers can be developed to better the code quality.

4.4. Frontend SPA Web App

Similar to the backend server, the frontend web app also uses NodeJs as the runtime environment and Yarn as the package manager. The full list of dependencies can be found in the *Frontend package.json* file in the Appendix.

As for the frontend single page web application itself, React is used as the main development framework. First, the barebone structure of a React project is created using the official preferred method through the use of the command

```
npx create-react-app frontend
```

The frontend project can then be populated with React different components. The components are divided into 4 main categories:

- Container Components
- Presentational Components
- Route Components
- Service Components

4.4.1. Container Components

Container components contain those that are used to wrap around other components to provide useful context data such as authentication and authorization data. In this system, only one container component is present, and it is the AuthorizationContainer. This is used to provide authorization status for all other components in the web application and is wrap around the entire web application. The implementation for this component can be found at src/container-component/AuthorizationContainer.tsx

4.4.2. Presentational Components

Presentational components are those that are used in other components to display certain similarly repeating components. These acts as the blueprints for UI components and can be created to have different values with the same design. The presentational components include, but are not limited to Buttons, Inputs, Dialogs, Forms and Modals.

Modals are used as the UI components to represent the objects in the system. This includes:

- Thesis
- Deployment
- Service
- Secret
- ConfigMap

Forms are used as the UI components for inputting objects into the system. This includes:

- Thesis
- Namespace

- Deployment
- Service
- Secret
- ConfigMap
- PersistentVolume
- PersistentVolumeClaim

4.4.3. Route Components

Route components contains the root components for each URL route in the SPA web app. These includes the components for every page in the web application. These are split into 2 main categories: **Home** and **Authorization**.

Home components contains the components for the homepage of the web app as well as other management pages such as thesis owner dashboard and admin dashboard.

Authorization components contains the components for all pages related to the login and signup process. This includes the landing page, the login page, the signup and forgot password pages.

4.4.4. Service Components

Service components contains all of the utility services for the SPA. This includes the GraphQL API calls, the context module, and other minor utilities. The most important module among these is no doubt the GraphQL API module, responsible for defining the API response body desired by the UI components. This helps reduce the excess information provided by the GraphQL API and only get the data required for each query.

CHAPTER 5

Experiment & Result

5.1. Experiment setup

The experiment is setup locally at the development machine used for this system, using the source code at the time of writing. As mentioned above, the source code can be found at https://github.com/Nitaray/thesis-management-system

The local machine is running on an 8th generation Intel i7 CPU with 32GB of RAM. This is not majorly impactful to the performance since there is a bottle neck on the amount of resource given to the Kubernetes cluster.

The backend server and frontend web application are deployed on localhost (127.0.0.1) using the npm script defined in the corresponding package.json files.

The Kubernetes cluster is running on a Minikube client, with the default namespace already populated with the PostgreSQL database container for the server.

The minikube cluster on the local machine is given the default 2 cpu cores and 2Gi of memory. This limits how fast and how many deployments can simultaneously run.

Several existing software systems of different complexity are also used to demonstrate the software redeployability of the thesis management system. These include:

- System A: 3 separate software deployed in tandem, with 1 database and 2 webservers, both using the same database.
- System B: A single websever using an offsite external database on the cloud.
- System C: A simple webserver serving static pages.

The structures of the systems are shown in the figures below

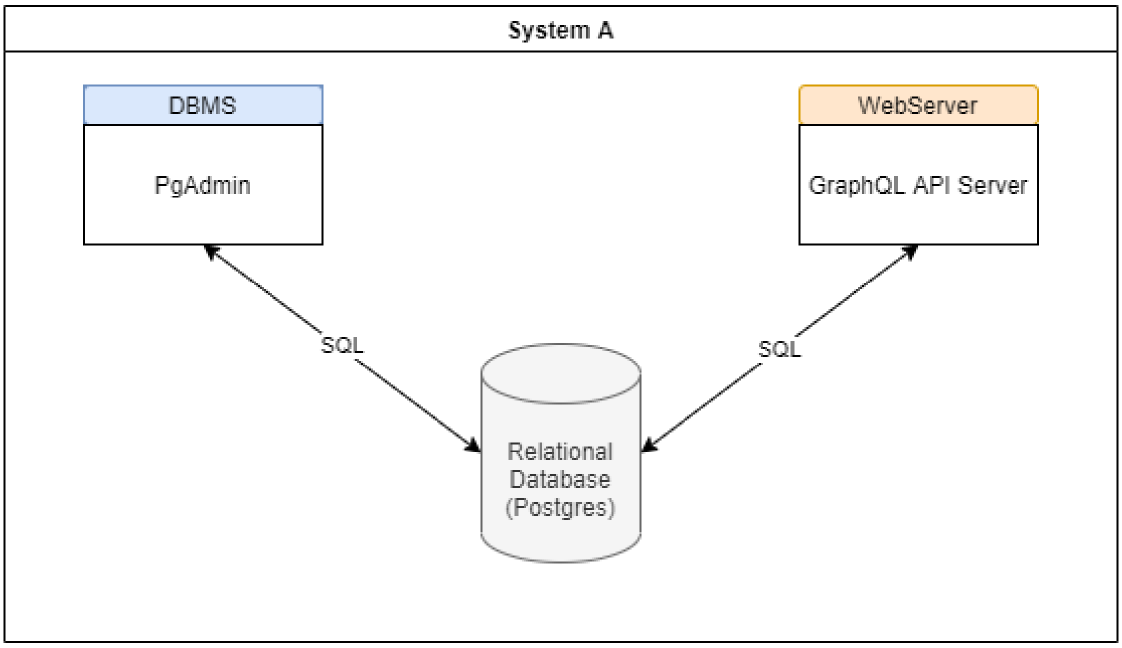


Figure 14. System A Structure

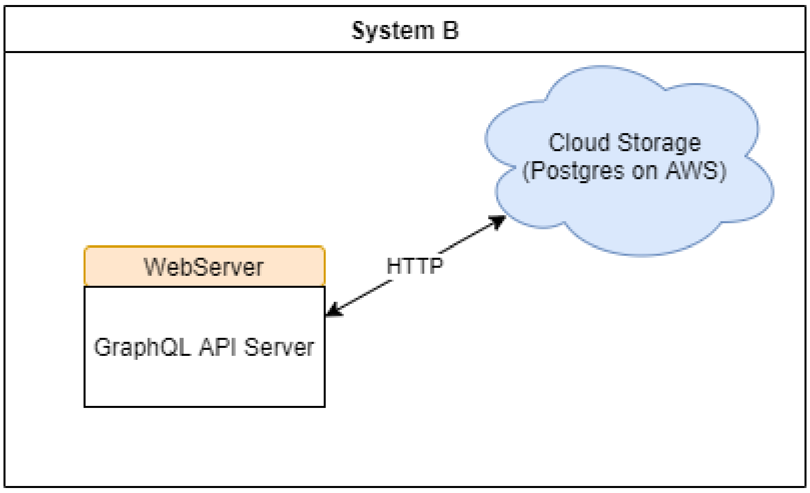


Figure 15. System B Structure

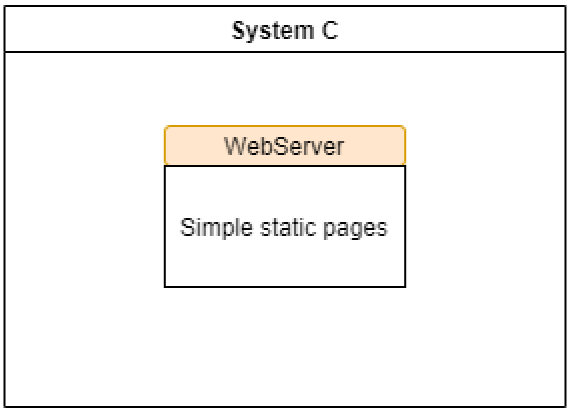


Figure 16. System C Structure

These systems were added in association with their own namespaces and are parts of separate theses using the thesis management system. All Kubernetes resource definition were mediated by the GraphQL API server developed.

5.2. Experiment result

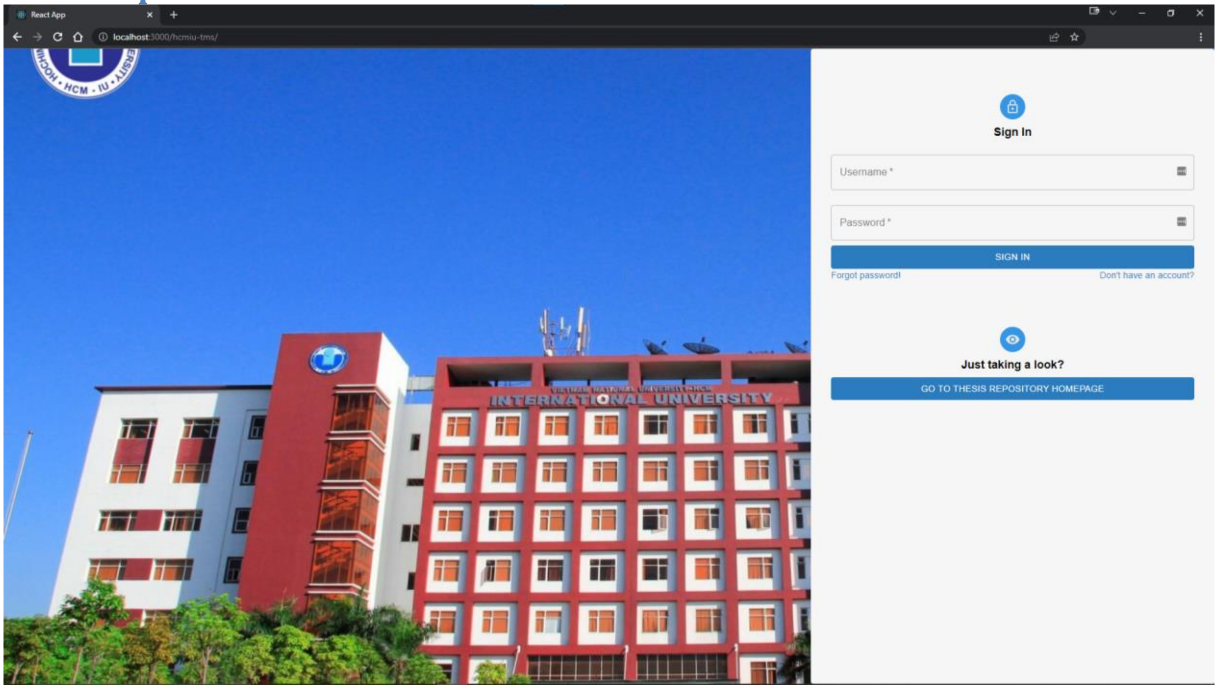
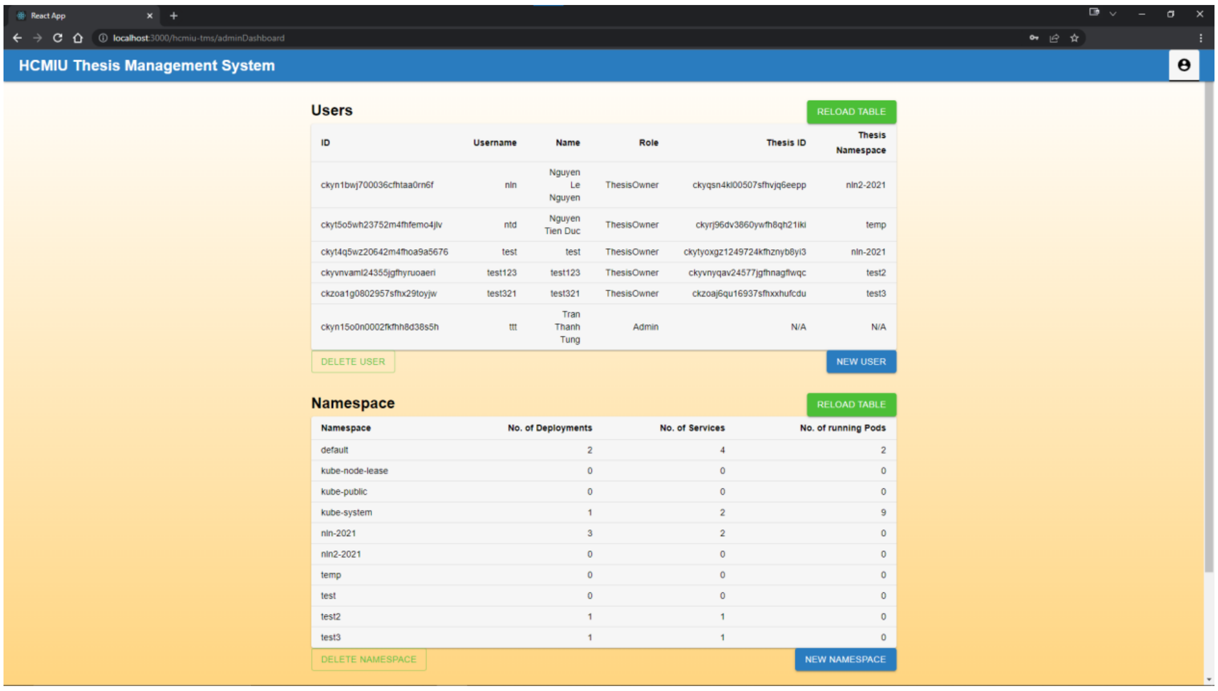


Figure 17. Landing Page



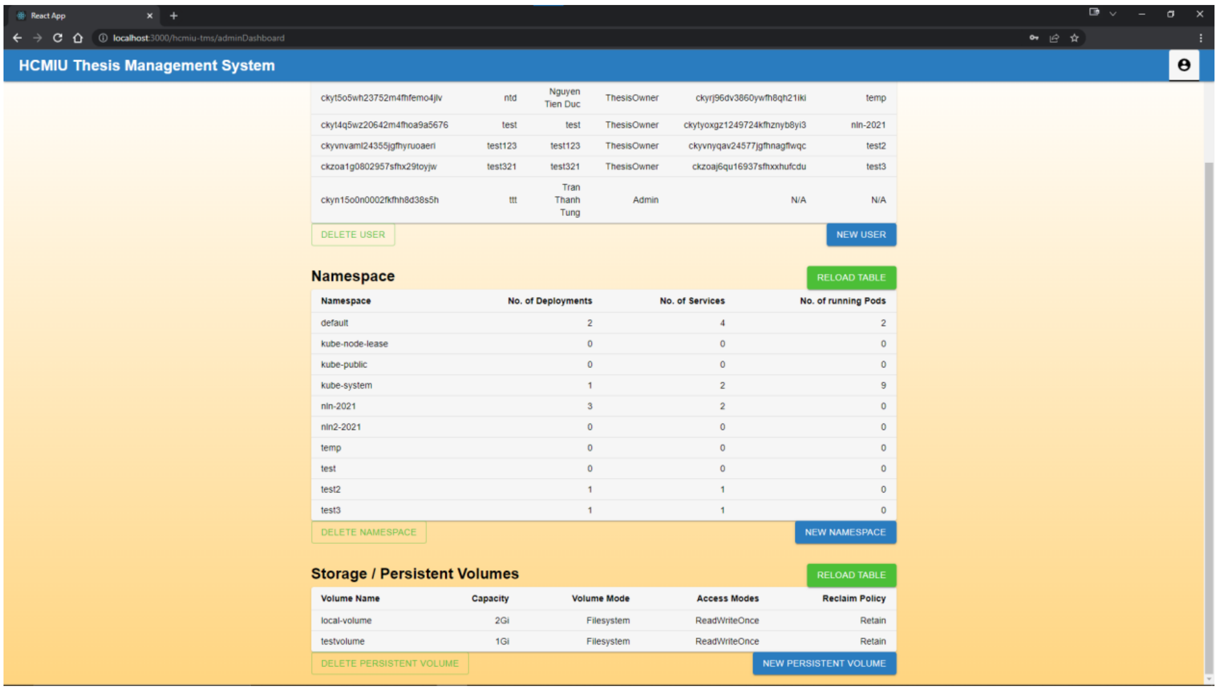
Users

ID	Username	Name	Role	Thesis ID	Thesis Namespace
ckyn1bwj700036cfhtaa0rn6f	nln	Nguyen Le Nguyen	ThesisOwner	ckyqsn4kl00507sfhvjq6eepp	nln2-2021
ckyt5o5wh23752m4fhfemo4jlv	ntd	Nguyen Tien Duc	ThesisOwner	ckyrj96dv3860ywfh8qh21iki	temp
ckyt4q5wz20642m4fhoa9a5676	test	test	ThesisOwner	ckytyoxgz1249724kfhznyb8yi3	nln-2021
ckyvnvaml24355jgfhyruoaeri	test123	test123	ThesisOwner	ckyvnyqav24577jgfhnagflwqc	test2
ckzoa1g0802957sfhx29toyjw	test321	test321	ThesisOwner	ckzoaj6qu16937sfhxxhufcdu	test3
ckyn15o0n0002fkfhh8d38s5h	ttt	Tran Thanh Tung	Admin	N/A	N/A

Namespace

Namespace	No. of Deployments	No. of Services	No. of running Pods
default	2	4	2
kube-node-lease	0	0	0
kube-public	0	0	0
kube-system	1	2	9
nln-2021	3	2	0
nln2-2021	0	0	0
temp	0	0	0
test	0	0	0
test2	1	1	0
test3	1	1	0

Figure 18. Admin Dashboard (Pt.1)



ckyt5o5wh23752m4fhfemo4jlv	ntd	Nguyen Tien Duc	ThesisOwner	ckyrj96dv3860ywfh8qh21iki	temp
ckyt4q5wz20642m4fhoa9a5676	test	test	ThesisOwner	ckytyoxgz1249724kfhznyb8yi3	nln-2021
ckyvnvaml24355jgfhyruoaeri	test123	test123	ThesisOwner	ckyvnyqav24577jgfhnagflwqc	test2
ckzoa1g0802957sfhx29toyjw	test321	test321	ThesisOwner	ckzoaj6qu16937sfhxxhufcdu	test3
ckyn15o0n0002fkfhh8d38s5h	ttt	Tran Thanh Tung	Admin	N/A	N/A

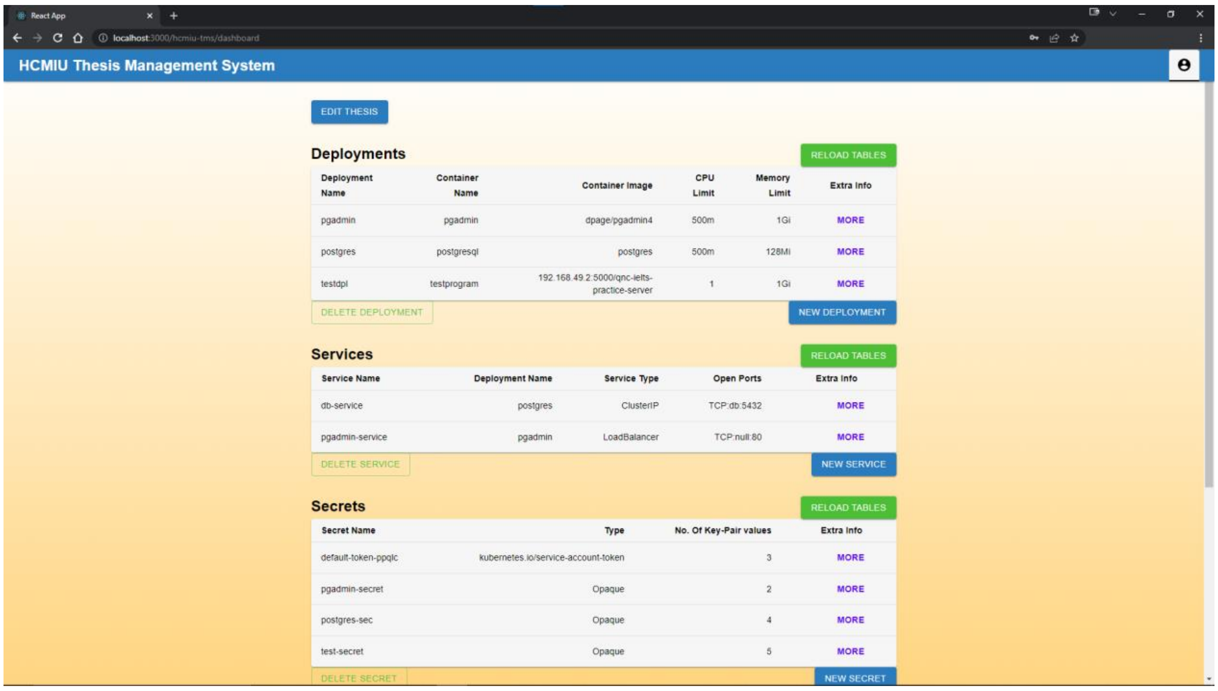
Namespace

Namespace	No. of Deployments	No. of Services	No. of running Pods
default	2	4	2
kube-node-lease	0	0	0
kube-public	0	0	0
kube-system	1	2	9
nln-2021	3	2	0
nln2-2021	0	0	0
temp	0	0	0
test	0	0	0
test2	1	1	0
test3	1	1	0

Storage / Persistent Volumes

Volume Name	Capacity	Volume Mode	Access Modes	Reclaim Policy
local-volume	2Gi	Filesystem	ReadWriteOnce	Retain
testvolume	1Gi	Filesystem	ReadWriteOnce	Retain

Figure 19. Admin Dashboard (Pt.2)



Deployments

Deployment Name	Container Name	Container Image	CPU Limit	Memory Limit	Extra Info
pgadmin	pgadmin	dpage/pgadmin4	500m	1Gi	MORE
postgres	postgresql	postgres	500m	128Mi	MORE
testdpl	testprogram	192.168.49.2:5000/qnc-ielts-practice-server	1	1Gi	MORE

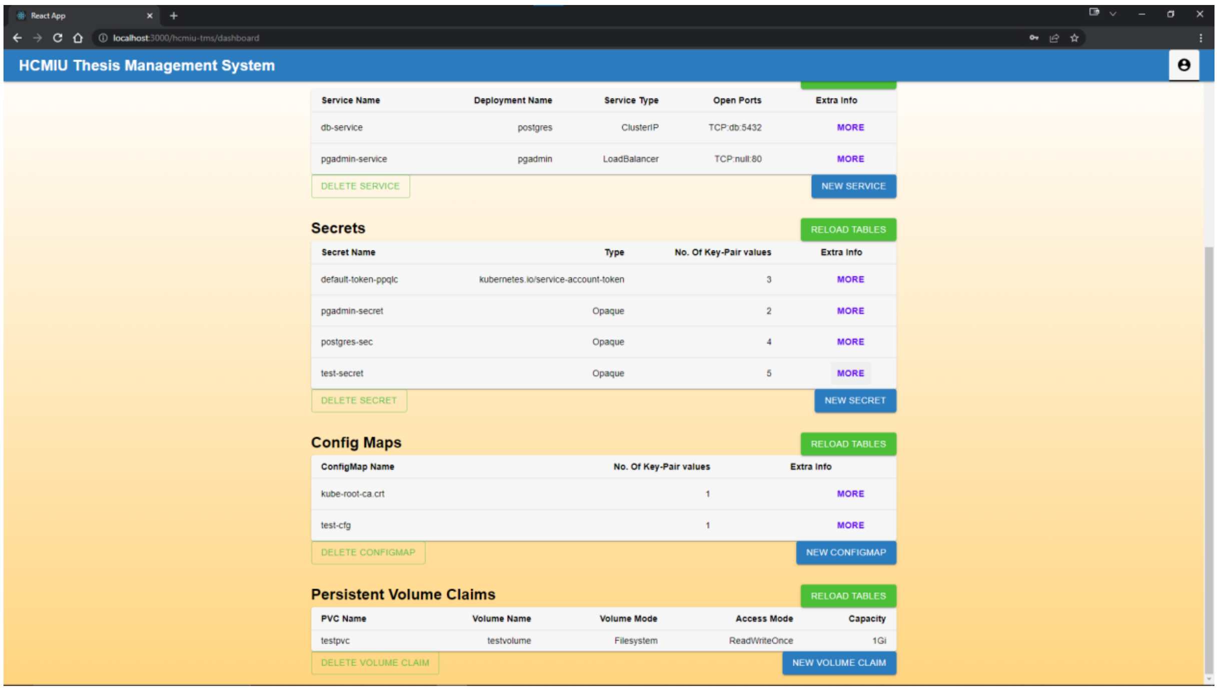
Services

Service Name	Deployment Name	Service Type	Open Ports	Extra Info
db-service	postgres	ClusterIP	TCP:db:5432	MORE
pgadmin-service	pgadmin	LoadBalancer	TCP:null:80	MORE

Secrets

Secret Name	Type	No. Of Key-Pair values	Extra Info
default-token-ppqlc	kubernetes.io/service-account-token	3	MORE
pgadmin-secret	Opaque	2	MORE
postgres-sec	Opaque	4	MORE
test-secret	Opaque	5	MORE

Figure 20. Thesis Owner Dashboard (Pt.1)


React App
localhost:3000/hcmiu-tms/dashboard
HCMIU Thesis Management System

Service Name	Deployment Name	Service Type	Open Ports	Extra Info
db-service	postgres	ClusterIP	TCP:db:5432	MORE
pgadmin-service	pgadmin	LoadBalancer	TCP:null:80	MORE

DELETE SERVICE NEW SERVICE

Secrets

RELOAD TABLES

Secret Name	Type	No. Of Key-Pair values	Extra Info
default-token-ppqlc	kubernetes.io/service-account-token	3	MORE
pgadmin-secret	Opaque	2	MORE
postgres-sec	Opaque	4	MORE
test-secret	Opaque	5	MORE

DELETE SECRET NEW SECRET

Config Maps

RELOAD TABLES

ConfigMap Name	No. Of Key-Pair values	Extra Info
kube-root-ca.crt	1	MORE
test-cfg	1	MORE

DELETE CONFIGMAP NEW CONFIGMAP

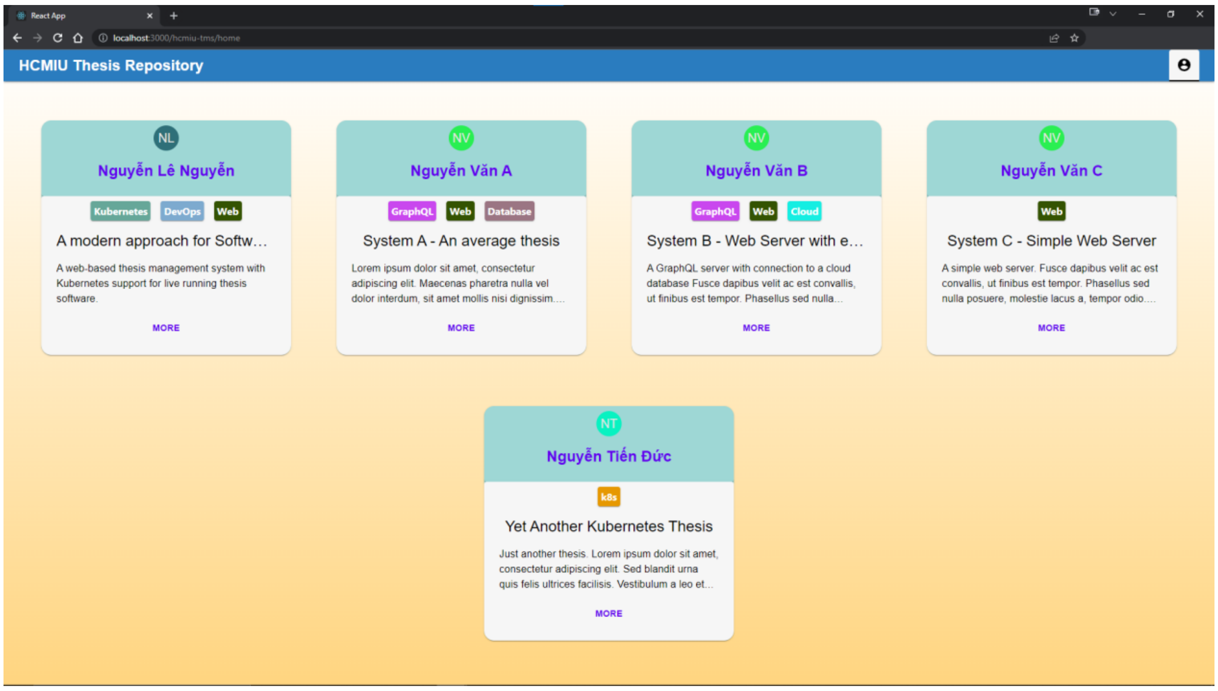
Persistent Volume Claims

RELOAD TABLES

PVC Name	Volume Name	Volume Mode	Access Mode	Capacity
testpvc	testvolume	Filesystem	ReadWriteOnce	1Gi

DELETE VOLUME CLAIM NEW VOLUME CLAIM

Figure 21. Thesis Owner Dashboard (Pt.2)



NL
Nguyễn Lê Nguyễn
Kubernetes DevOps Web
A modern approach for Softw…
A web-based thesis management system with Kubernetes support for live running thesis software.
MORE

NV
Nguyễn Văn A
GraphQL Web Database
System A - An average thesis
Lorem ipsum dolor sit amet, consectetur adipiscing elit. Maecenas pharetra nulla vel dolor interdum, sit amet mollis nisi dignissim....
MORE

NV
Nguyễn Văn B
GraphQL Web Cloud
System B - Web Server with e…
A GraphQL server with connection to a cloud database Fusce dapibus velit ac est convallis, ut finibus est tempor. Phasellus sed nulla...
MORE

NV
Nguyễn Văn C
Web
System C - Simple Web Server
A simple web server. Fusce dapibus velit ac est convallis, ut finibus est tempor. Phasellus sed nulla posuere, molestie lacus a, tempor odio....
MORE

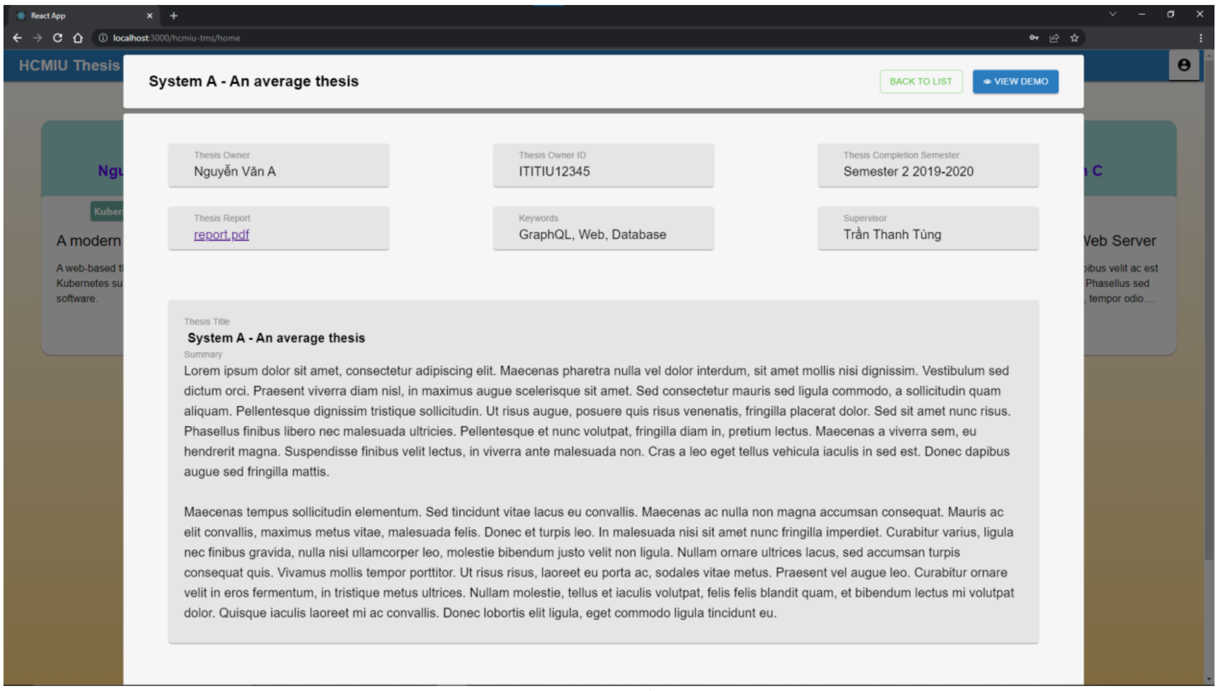
NT
Nguyễn Tiến Đức
k8s
Yet Another Kubernetes Thesis
Just another thesis. Lorem ipsum dolor sit amet, consectetur adipiscing elit. Sed blandit urna quis felis ultrices facilisis. Vestibulum a leo et...
MORE

Figure 22. Thesis Viewing Homepage

Some of the main pages supported by the thesis management system is shown in the figures above. There are a lot more dialogs and forms that are not shown as the main purpose of this experiment is not to show the design of the user interface. The primary purpose of this experiment is to evaluate the software redeployability capability of the thesis management system using Kubernetes.

To evaluate the capability of the system, the previous 3 systems are used. The three systems are associated with the 3 theses with the corresponding titles seen in Figure 22. Clicking on the

MORE button will show a modal with the detail information of each theses and their namespace resources.



System A - An average thesis

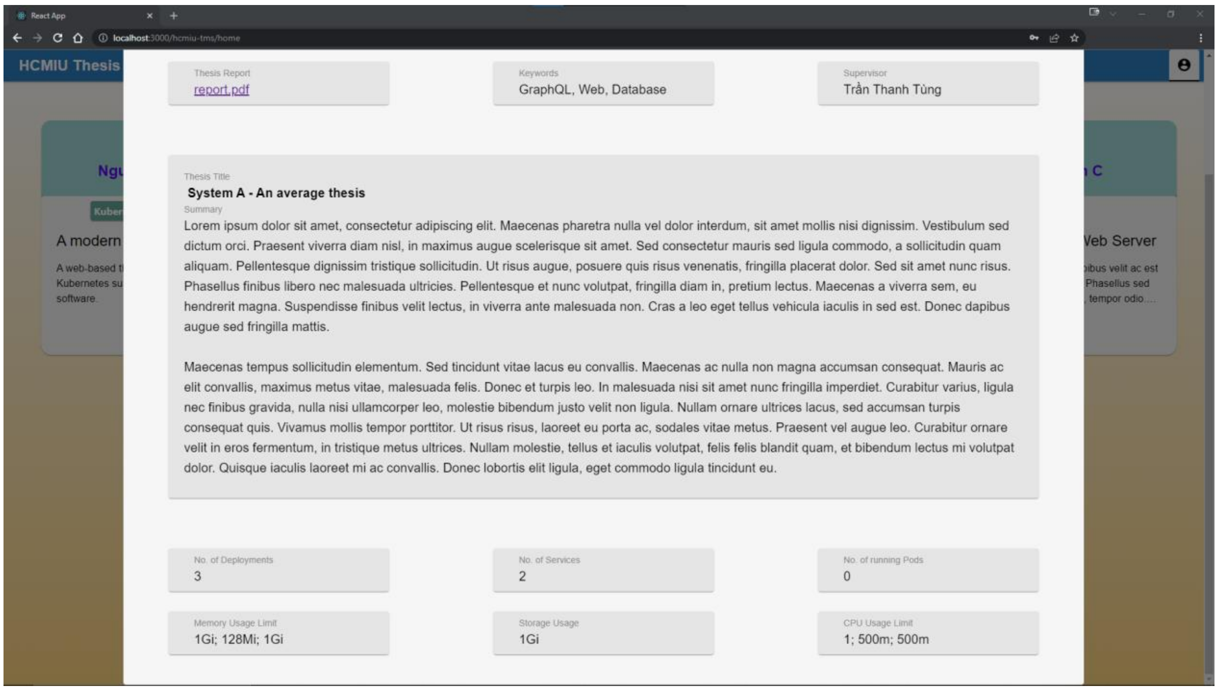
Thesis Owner Nguyễn Văn A	Thesis Owner ID ITITIU12345	Thesis Completion Semester Semester 2 2019-2020
Thesis Report report.pdf	Keywords GraphQL, Web, Database	Supervisor Trần Thanh Tùng

Thesis Title
System A - An average thesis
Summary
Lorem ipsum dolor sit amet, consectetur adipiscing elit. Maecenas pharetra nulla vel dolor interdum, sit amet mollis nisi dignissim. Vestibulum sed dictum orci. Praesent viverra diam nisl, in maximus augue scelerisque sit amet. Sed consectetur mauris sed ligula commodo, a sollicitudin quam aliquam. Pellentesque dignissim tristique sollicitudin. Ut risus augue, posuere quis risus venenatis, fringilla placerat dolor. Sed sit amet nunc risus. Phasellus finibus libero nec malesuada ultricies. Pellentesque et nunc volutpat, fringilla diam in, pretium lectus. Maecenas a viverra sem, eu hendrerit magna. Suspendisse finibus velit lectus, in viverra ante malesuada non. Cras a leo eget tellus vehicula iaculis in sed est. Donec dapibus augue sed fringilla mattis.

Maecenas tempus sollicitudin elementum. Sed tincidunt vitae lacus eu convallis. Maecenas ac nulla non magna accumsan consequat. Mauris ac elit convallis, maximus metus vitae, malesuada felis. Donec et turpis leo. In malesuada nisi sit amet nunc fringilla imperdiet. Curabitur varius, ligula nec finibus gravida, nulla nisi ullamcorper leo, molestie bibendum justo velit non ligula. Nullam ornare ultrices lacus, sed accumsan turpis consequat quis. Vivamus mollis tempor porttitor. Ut risus risus, laoreet eu porta ac, sodales vitae metus. Praesent vel augue leo. Curabitur ornare velit in eros fermentum, in tristique metus ultrices. Nullam molestie, tellus et iaculis volutpat, felis felis blandit quam, et bibendum lectus mi volutpat dolor. Quisque iaculis laoreet mi ac convallis. Donec lobortis elit ligula, eget commodo ligula tincidunt eu.

Figure 23. System A Information (Pt.1)

Kubernetes resource information such as number of deployments and resource limits are shown at the bottom of the modal as shown by Figure 24.



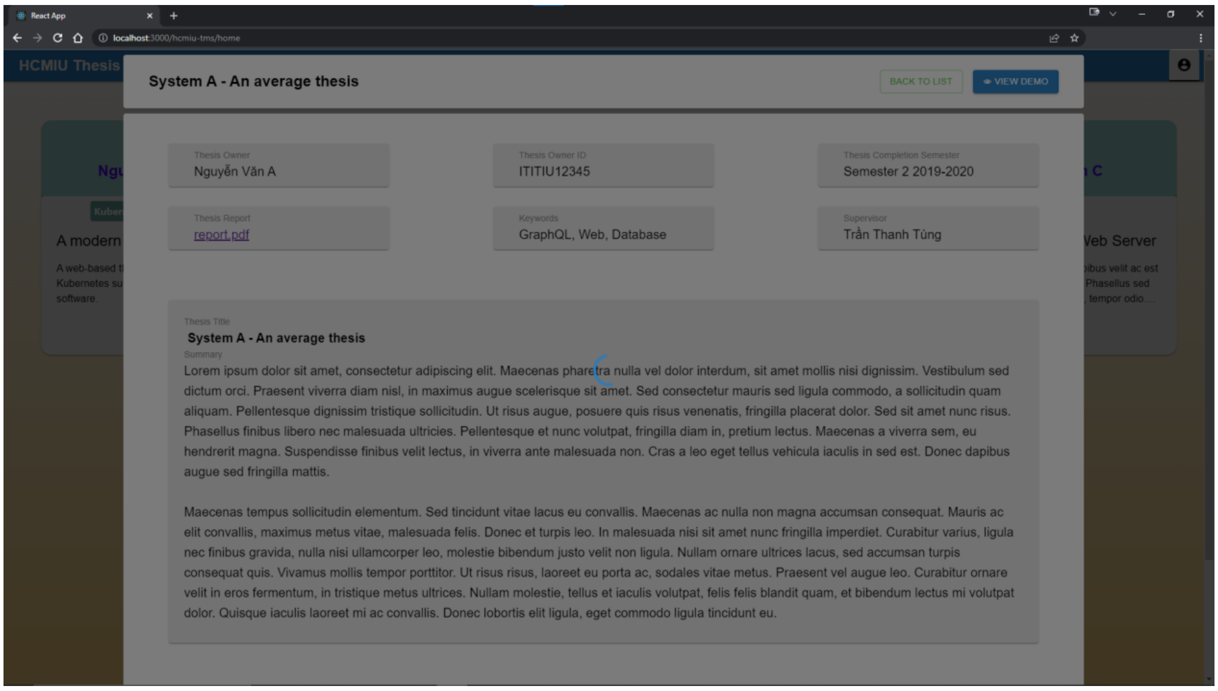
Thesis Report report.pdf	Keywords GraphQL, Web, Database	Supervisor Trần Thanh Tùng

Thesis Title
System A - An average thesis
Summary
Lorem ipsum dolor sit amet, consectetur adipiscing elit. Maecenas pharetra nulla vel dolor interdum, sit amet mollis nisi dignissim. Vestibulum sed dictum orci. Praesent viverra diam nisl, in maximus augue scelerisque sit amet. Sed consectetur mauris sed ligula commodo, a sollicitudin quam aliquam. Pellentesque dignissim tristique sollicitudin. Ut risus augue, posuere quis risus venenatis, fringilla placerat dolor. Sed sit amet nunc risus. Phasellus finibus libero nec malesuada ultricies. Pellentesque et nunc volutpat, fringilla diam in, pretium lectus. Maecenas a viverra sem, eu hendrerit magna. Suspendisse finibus velit lectus, in viverra ante malesuada non. Cras a leo eget tellus vehicula iaculis in sed est. Donec dapibus augue sed fringilla mattis.

Maecenas tempus sollicitudin elementum. Sed tincidunt vitae lacus eu convallis. Maecenas ac nulla non magna accumsan consequat. Mauris ac elit convallis, maximus metus vitae, malesuada felis. Donec et turpis leo. In malesuada nisi sit amet nunc fringilla imperdiet. Curabitur varius, ligula nec finibus gravida, nulla nisi ullamcorper leo, molestie bibendum justo velit non ligula. Nullam ornare ultrices lacus, sed accumsan turpis consequat quis. Vivamus mollis tempor porttitor. Ut risus risus, laoreet eu porta ac, sodales vitae metus. Praesent vel augue leo. Curabitur ornare velit in eros fermentum, in tristique metus ultrices. Nullam molestie, tellus et iaculis volutpat, felis felis blandit quam, et bibendum lectus mi volutpat dolor. Quisque iaculis laoreet mi ac convallis. Donec lobortis elit ligula, eget commodo ligula tincidunt eu.

No. of Deployments 3	No. of Services 2	No. of running Pods 0
Memory Usage Limit 1Gi; 128Mi; 1Gi	Storage Usage 1Gi	CPU Usage Limit 1; 500m; 500m

Figure 24. System A Information (Pt.2)



System A - An average thesis

Thesis Owner Nguyễn Văn A	Thesis Owner ID ITITIU12345	Thesis Completion Semester Semester 2 2019-2020
Thesis Report report.pdf	Keywords GraphQL, Web, Database	Supervisor Trần Thanh Tùng

Thesis Title
System A - An average thesis
Summary
Lorem ipsum dolor sit amet, consectetur adipiscing elit. Maecenas pharetra nulla vel dolor interdum, sit amet mollis nisi dignissim. Vestibulum sed dictum orci. Praesent viverra diam nisl, in maximus augue scelerisque sit amet. Sed consectetur mauris sed ligula commodo, a sollicitudin quam aliquam. Pellentesque dignissim tristique sollicitudin. Ut risus augue, posuere quis risus venenatis, fringilla placerat dolor. Sed sit amet nunc risus. Phasellus finibus libero nec malesuada ultricies. Pellentesque et nunc volutpat, fringilla diam in, pretium lectus. Maecenas a viverra sem, eu hendrerit magna. Suspendisse finibus velit lectus, in viverra ante malesuada non. Cras a leo eget tellus vehicula iaculis in sed est. Donec dapibus augue sed fringilla mattis.

Maecenas tempus sollicitudin elementum. Sed tincidunt vitae lacus eu convallis. Maecenas ac nulla non magna accumsan consequat. Mauris ac elit convallis, maximus metus vitae, malesuada felis. Donec et turpis leo. In malesuada nisi sit amet nunc fringilla imperdiet. Curabitur varius, ligula nec finibus gravida, nulla nisi ullamcorper leo, molestie bibendum justo velit non ligula. Nullam ornare ultrices lacus, sed accumsan turpis consequat quis. Vivamus mollis tempor porttitor. Ut risus risus, laoreet eu porta ac, sodales vitae metus. Praesent vel augue leo. Curabitur ornare velit in eros fermentum, in tristique metus ultrices. Nullam molestie, tellus et iaculis volutpat, felis felis blandit quam, et bibendum lectus mi volutpat dolor. Quisque iaculis laoreet mi ac convallis. Donec lobortis elit ligula, eget commodo ligula tincidunt eu.

Figure 25. System A - Clicking View Demo

For thesis with demo viewing capability, click the view demo button will enter a loading state, waiting for the backend server to start up the deployments in the Kubernetes server and forwarding the correct path to access them.

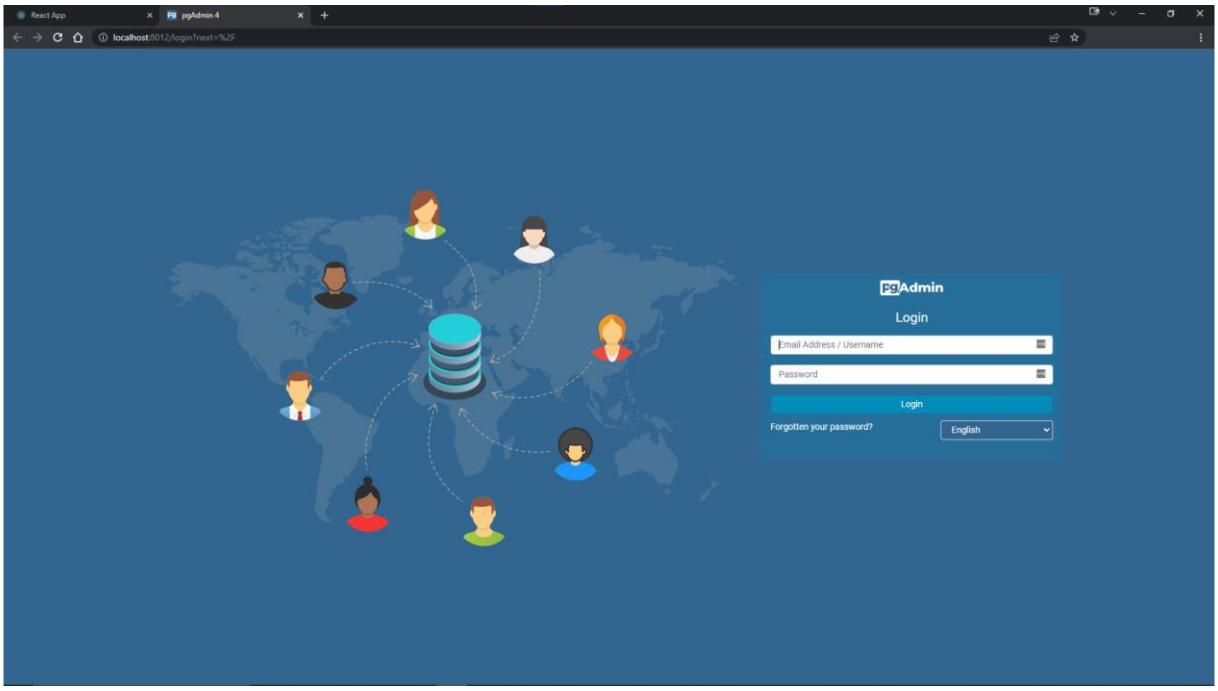
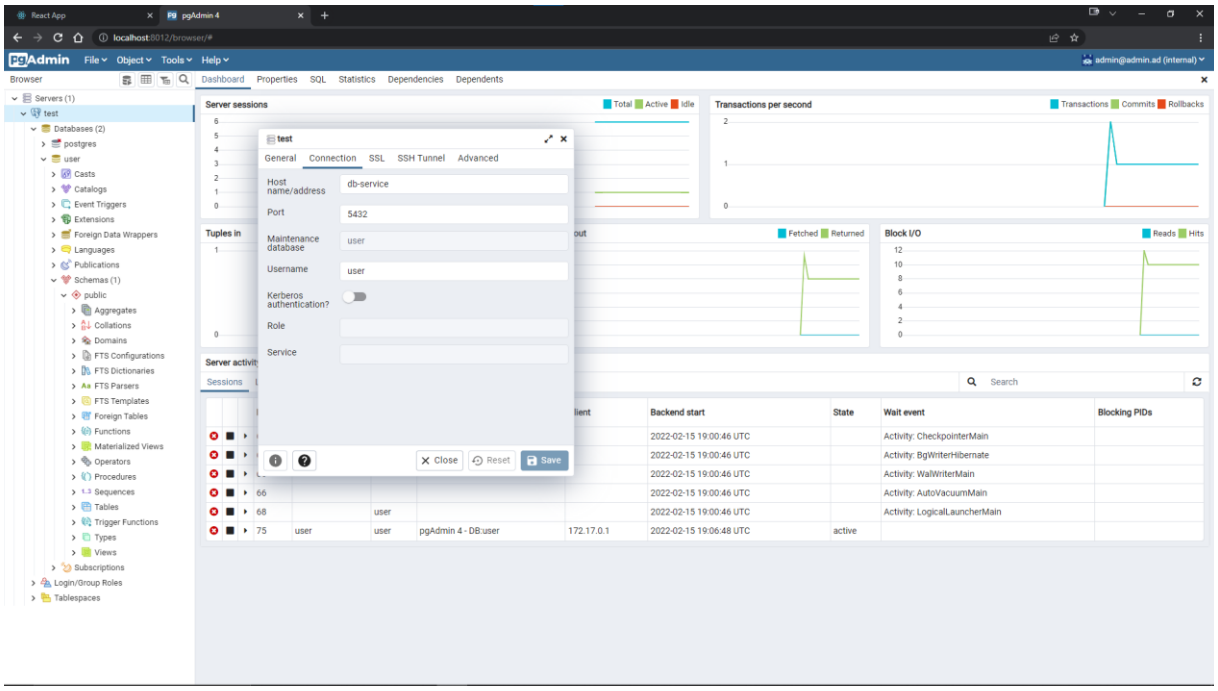


Figure 26. System A - Demo Opened

For system A, the thesis's front-facing service is set to the pgAdmin web server. This is in no way obligated and can be set to any service running in the namespace.

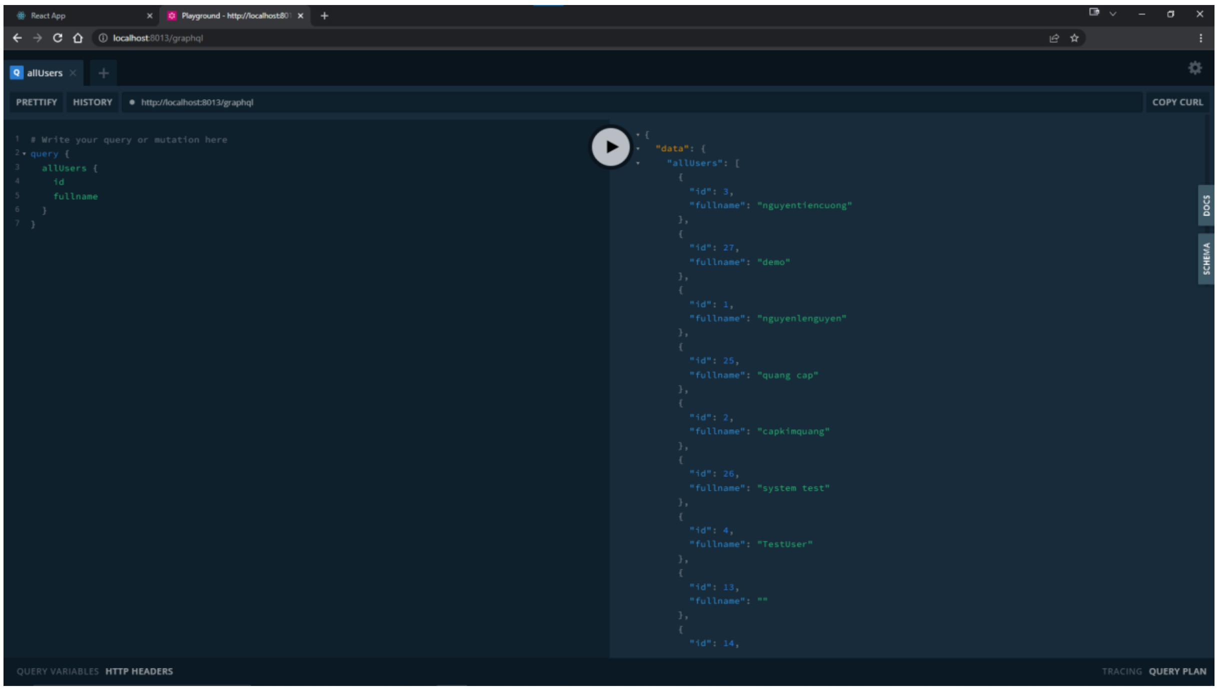


Host name/address	db-service
Port	5432
Maintenance database	user
Username	user
Kerberos authentication?	
Role	
Service	

							Backend start	State	Wait event	Blocking PIDs
							2022-02-15 19:00:46 UTC		Activity: CheckpointerMain	
							2022-02-15 19:00:46 UTC		Activity: BgWriterHibernate	
							2022-02-15 19:00:46 UTC		Activity: WalWriterMain	
		66					2022-02-15 19:00:46 UTC		Activity: AutoVacuumMain	
		68		user			2022-02-15 19:00:46 UTC		Activity: LogicalLauncherMain	
		75	user	user	pgAdmin 4 - DB:user	172.17.0.1	2022-02-15 19:06:48 UTC	active		

Figure 27. System A - Demo connect internal service

After logging into the pgAdmin using the credentials defined by the secrets in the namespace, a simple connection to the internal database using the service name as the host confirms that the pgAdmin webserver is indeed connected to the internal database and is receiving connection data as shown by the graphs on the right.

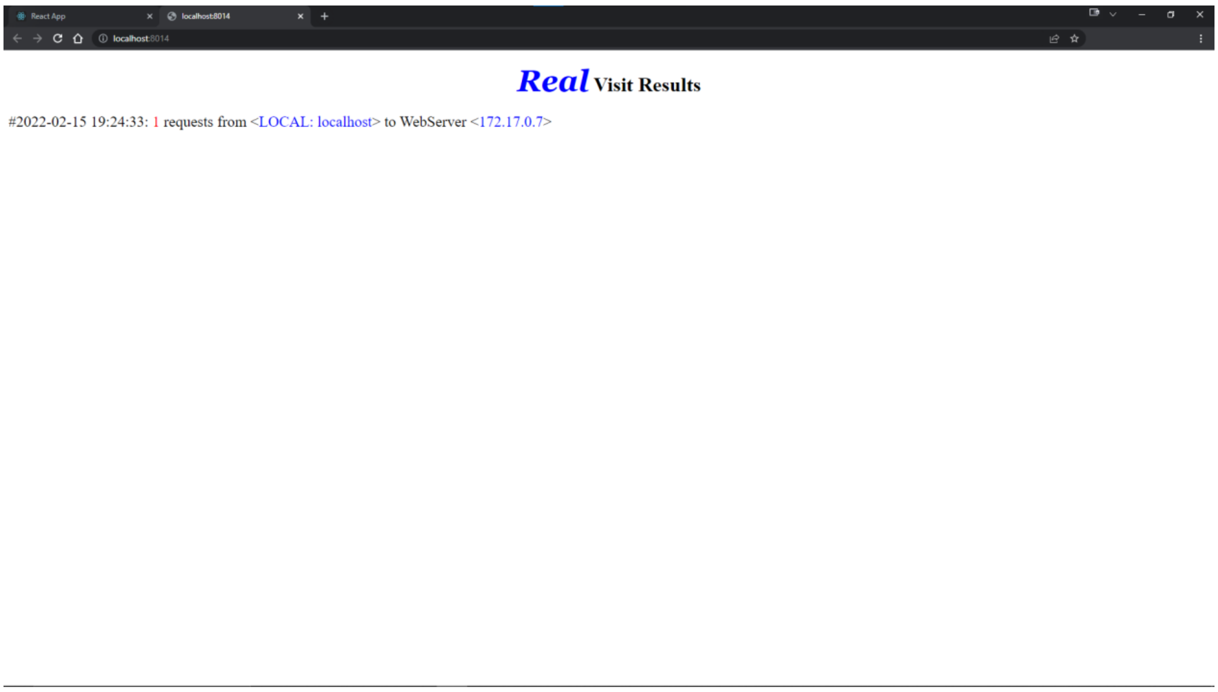


```
# Write your query or mutation here
query {
  allUsers {
    id
    fullname
  }
}
```

```
{
  "data": {
    "allUsers": [
      {
        "id": 3,
        "fullname": "nguyentiencuong"
      },
      {
        "id": 27,
        "fullname": "demo"
      },
      {
        "id": 1,
        "fullname": "nguyenlenguyen"
      },
      {
        "id": 25,
        "fullname": "quang cap"
      },
      {
        "id": 2,
        "fullname": "capkimquang"
      },
      {
        "id": 26,
        "fullname": "system test"
      },
      {
        "id": 4,
        "fullname": "TestUser"
      },
      {
        "id": 13,
        "fullname": ""
      },
      {
        "id": 14,
```

Figure 28. System B - Demo Opened

For system B, the demo opened a GraphQL API query interface, which is served by the webserver running on the Kubernetes cluster. This webserver however draws data from a Postgres database instance on AWS cloud storage. The query successfully returns the data present in the database. This can be verified via the API hosted at https://qnc-ielts-practice.herokuapp.com/graphql.



Real Visit Results

#2022-02-15 19:24:33: 1 requests from <LOCAL: localhost> to WebServer <172.17.0.7>

Figure 29. System C - Demo opened

System C is only a simple webserver hosted on port 80 and returns a static webpage showing the client IP as well as the server IP. In all of the 3 systems, system C is the most simplistic.

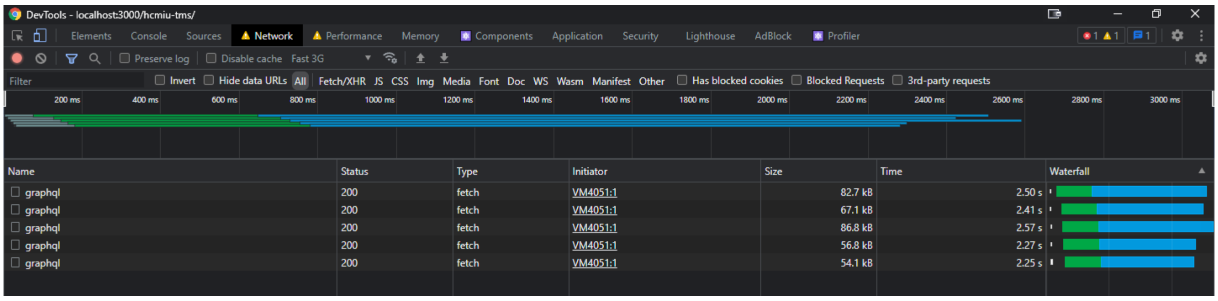
5.3. Evaluation

The evaluation of the experiment will be based on 2 parts: the completion of the main objectives and the performance criteria defined in section 3.6.

As for the main objectives, all of them are completed to satisfaction, including

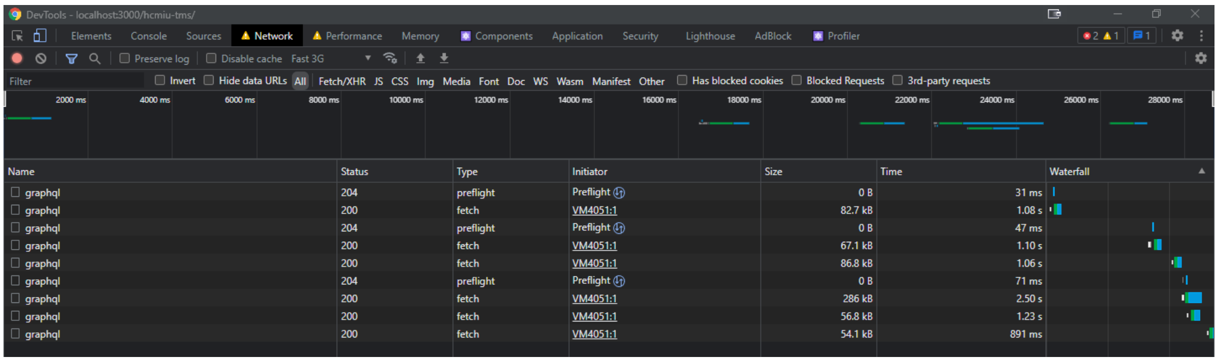
- Thesis information management via web application (Administrative and non-administrative).
- Kubernetes deployment and redeployment of Docker-containerized thesis software.
- Thesis software demo viewing using automatic Kubernetes deployment.

As for the performance criteria, to fairly measure the website's performance, Google Chrome's builtin network throttle is used. With the throttle set to "Mid-tier mobile" network performance, the website load time still exceeds the criteria expectation.



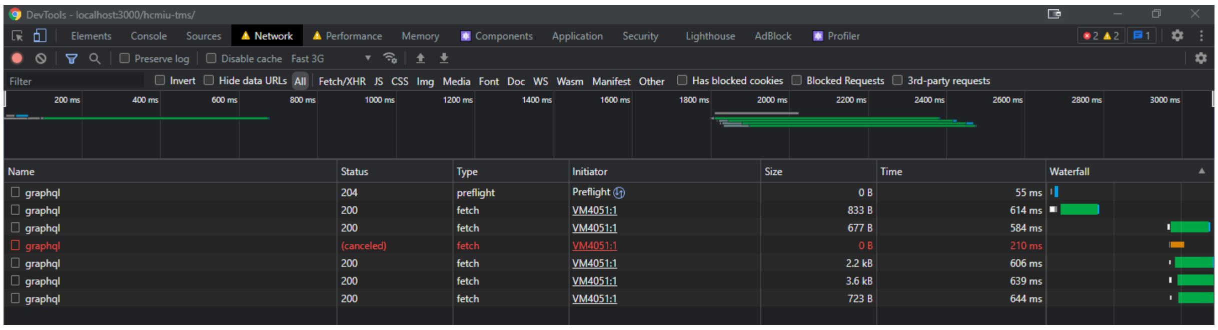
Name	Status	Type	Initiator	Size	Time
graphql	200	fetch	VM4051:1	82.7 kB	2.50 s
graphql	200	fetch	VM4051:1	67.1 kB	2.41 s
graphql	200	fetch	VM4051:1	86.8 kB	2.57 s
graphql	200	fetch	VM4051:1	56.8 kB	2.27 s
graphql	200	fetch	VM4051:1	54.1 kB	2.25 s

Figure 30. Performance loading home page



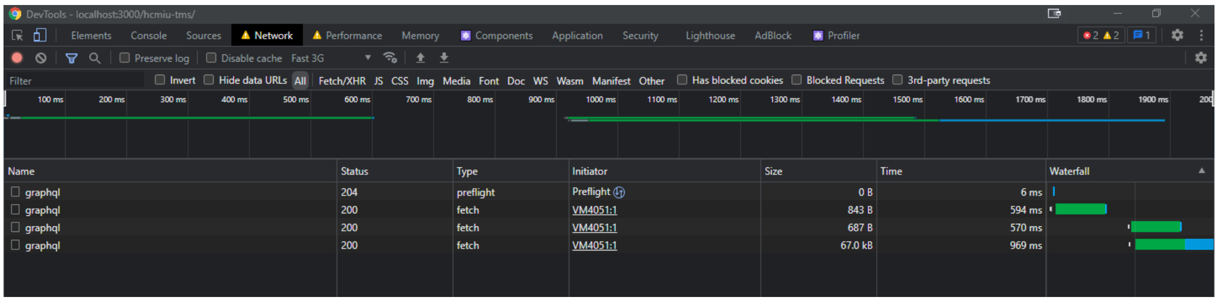
Name	Status	Type	Initiator	Size	Time
graphql	204	preflight	Preflight	0 B	31 ms
graphql	200	fetch	VM4051:1	82.7 kB	1.08 s
graphql	204	preflight	Preflight	0 B	47 ms
graphql	200	fetch	VM4051:1	67.1 kB	1.10 s
graphql	200	fetch	VM4051:1	86.8 kB	1.06 s
graphql	204	preflight	Preflight	0 B	71 ms
graphql	200	fetch	VM4051:1	286 kB	2.50 s
graphql	200	fetch	VM4051:1	56.8 kB	1.23 s
graphql	200	fetch	VM4051:1	54.1 kB	891 ms

Figure 31. Performance loading every thesis information



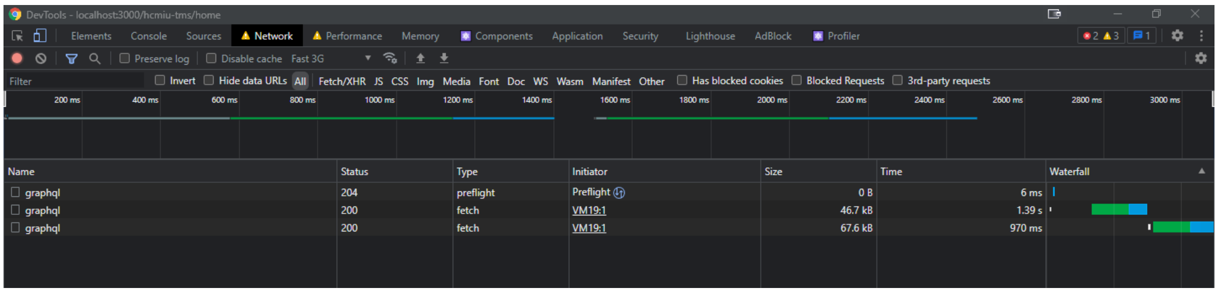
Name	Status	Type	Initiator	Size	Time
graphql	204	preflight	Preflight	0 B	55 ms
graphql	200	fetch	VM4051:1	833 B	614 ms
graphql	200	fetch	VM4051:1	677 B	584 ms
graphql	(canceled)	fetch	VM4051:1	0 B	210 ms
graphql	200	fetch	VM4051:1	2.2 kB	606 ms
graphql	200	fetch	VM4051:1	3.6 kB	639 ms
graphql	200	fetch	VM4051:1	723 B	644 ms

Figure 32. Performance logging in and loading admin dashboard



Name	Status	Type	Initiator	Size	Time
graphql	204	preflight	Preflight	0 B	6 ms
graphql	200	fetch	VM4051:1	843 B	594 ms
graphql	200	fetch	VM4051:1	687 B	570 ms
graphql	200	fetch	VM4051:1	67.0 kB	969 ms

Figure 33. Performance logging in and loading thesis owner dashboard

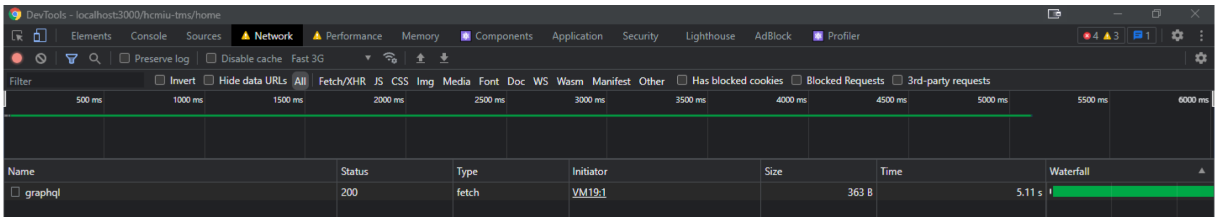

DevTools - localhost:3000/hcmiu-tms/home

Fast 3G

Name	Status	Type	Initiator	Size	Time
graphql	204	preflight	Preflight	0 B	6 ms
graphql	200	fetch	VM19:1	46.7 kB	1.39 s
graphql	200	fetch	VM19:1	67.6 kB	970 ms

Figure 34. Performance updating thesis information

As shown in the above figures, web pages load time is well within criteria expectation as well as in the acceptable threshold especially for the low network performance. This is thanks to the use of GraphQL, which significantly reduces the number of requests needed to be sent for retrieving the data on each page.


DevTools - localhost:3000/hcmiu-tms/home

Fast 3G

Name	Status	Type	Initiator	Size	Time
graphql	200	fetch	VM19:1	363 B	5.11 s

Figure 35. Performance retrieving thesis demo link

For the software demo and deployment, however, the time taken is higher. Since the resources designated to the Kubernetes cluster is limited, it takes around 10 – 20 seconds for complex systems such as System A to deploy. For simpler system such as System C, the time taken is much less, and is almost instant. The 5 seconds seen in Figure 35 is actually hard set by the server in order to ensure stability when deploying.

To reduce the time taken for deploying the software demos, the most practical solution is to allocate more resource to the Kubernetes cluster as well as using better hardware. Since this is a problem inherent in hardware performance, the final evaluation of the developed system is exceeding satisfactory.

CHAPTER 6

DISCUSSION

6.1. Discussion

Even though the current developed thesis management system exceeds the set criteria and satisfy all of the main objectives, it is not without its problems. These cons as well as the pros that this system design brings will be discussed in this section.

Starting with the pros, the developed system:

- Successfully demonstrate the capability of Kubernetes and Containerization technology in solving the software management problem.
- Provides functionalities no other thesis management system has provided until now.
- Lightweight, capable of running smoothly even for old-fashioned devices.

On the other hand, it also has some drawbacks:

- Only support software demo for web-based software, providing access to thesis software through a web link.
- Require system-dependant setups before hand in order to run properly.

Although the proposed thesis management system works well for its intended purposes, it is evident that the purposes are still lacking when compared to the larger software management problem. To overcome these shortcomings, the system design will need further refinements and developments, both of which will consume valuable time resource that unfortunately is not in the budget of this thesis.

6.2. Comparison

As mentioned in earlier chapters, there are few software management system and no thesis management system which can provide a satisfactory solution to the old thesis software maintenance problem proposed by this thesis. As such, as also mentioned in Chapter 2, there are few comparisons to the proposed thesis management system. The only prominent industry counterpart is KNative.

6.2.1. Compare to industry solutions

At first glance, Knative seems to solve the same software management and deployment problem using the same Kubernetes technology as the proposed thesis management system. However, this is a misconception since the two systems solve two very different problems. While Knative provides platform for managing software deployment on Kubernetes, Knative aims for only serverless, cloud-native applications. This combines with the fact that it is designed with large industry-scaled applications in mind, it would be overcomplicated and overwhelming for thesis students to learn and use this piece of software.

On the other hand, the proposed thesis management system aims at a wider range of applications, providing an on-premise application management system that is easy to use for thesis students as well as provide reviewability for suitable old thesis software. The proposed system drastically abstract away the complexity of dealing with a Kubernetes cluster to a user-friendly web interface and straight-forward functionalities.

In short, comparing Knative and the proposed system is analogous to comparing apples to pineapples. They sound the same but they are not the same and has very different characteristics.

6.2.2. Compare to education field solutions

As stated above, there are little to no direct comparisons to the proposed thesis management system in the education field. Most thesis management systems are just barebone Content Management Systems (CMSs), capable of keeping track of information and files related to the thesis, with no support whatsoever for thesis software. As such, there are no comparisons to be made in this section.

CHAPTER 7

CONCLUSION AND FUTURE WORK

7.1. Conclusion

In brief, this thesis proposed a thesis management system suitable for managing both the thesis information as well as the software that may come along with each thesis. The system combines state-of-the-art containerization technologies with modern web technologies to solve the software redeployability problem present in most old software-based theses.

For the proposed system, a backend server and frontend web application is developed along with configurations for a local Kubernetes cluster using Minikube. Both the backend and frontend is developed using Typescript as the main language and utilized modern frameworks and libraries to aid in the development process. The result is a feature-complete thesis management system that also provides capability for deploying software and viewing demos of web-based ones.

The thesis management system proposed, while manages to satisfy all of the planned requirements and demonstrate the feasibility of using Kubernetes to solve the thesis software management problem, still has room to be improved and extended upon. Extensions to this thesis can be in either depth, such as providing a more complete feature set and more tightly integrated with Kubernetes, or width, such as providing more functionalities for thesis information management and cover more types of theses, not just software-based.

7.2. Future works

As summarized in the previous section, the proposed thesis management system still has a lot of room to grow and for future theses to be based on. Some of the immediately visible future works focuses on expanding the range of support for different types of thesis as well as streamlining the automation of development and testing of thesis software. Specifically, some ideas for future works can be described as the following:

- Support for viewing demo of non web-based applications, or in other words, native applications.
- Support for more types of thesis than just software, such as research or case studies.
- Support for automated testing of deployed software. This can include prepared tests by the thesis owner, or testing methods in the development process such as A/B testing, etc.

REFERENCES

1. Docker Documentation. *Docker.* [Online] https://docs.docker.com/get-started/overview/.

2. What is a hypervisor? *Vmware.com.* [Online] Vmware. https://www.vmware.com/topics/glossary/content/hypervisor.html.

3. Does Docker run on Linux, MacOS and Windows. *Docker.com.* [Online] Docker. https://docs.docker.com/engine/faq/#does-docker-run-on-linux-macos-and-windows.

4. Docker Architecture. *Docker.com.* [Online] Docker. https://docs.docker.com/get-started/overview/#docker-architecture.

5. Docker Daemon. *Docker.com.* [Online] Docker. https://docs.docker.com/get-started/overview/#the-docker-daemon.

6. Docker Client. *Docker.com.* [Online] Docker. https://docs.docker.com/get-started/overview/#the-docker-client.

7. Docker Objects. *Docker.com.* [Online] Docker. https://docs.docker.com/get-started/overview/#docker-objects.

8. Docker Registry. *Docker.com.* [Online] https://docs.docker.com/get-started/overview/#docker-registries.

9. Jorge Castro, Duffie Cooley, Kat Cosgrove, Justin Garrison, Noah Kantrowitz, Bob Killen, Rey Lejano, Dan “POP” Papandrea, Jeffrey Sica, Davanum “Dims” Srinivas. Don't Panic: Kubernetes and Docker. *kubernetes.io.* [Online] Kubernetes. https://kubernetes.io/blog/2020/12/02/dont-panic-kubernetes-and-docker/#so-why-the-confusion-and-what-is-everyone-freaking-out-about.

10. Kubernetes. *kubernetes.io.* [Online] Kubernetes. https://kubernetes.io/.

11. Kubernetes Nodes. *kubernetes.io.* [Online] Kubernetes. https://kubernetes.io/docs/concepts/architecture/nodes/.

12. Control Plane - Node communication. *kubernetes.io.* [Online] Kubernetes. https://kubernetes.io/docs/concepts/architecture/control-plane-node-communication/.

13. Kubelet. *kubernetes.io.* [Online] Kubernetes. https://kubernetes.io/docs/reference/command-line-tools-reference/kubelet/.

14. Kubernetes Pods. *kubernetes.io.* [Online] Kubernetes. https://kubernetes.io/docs/concepts/workloads/pods/.

15. Kubernetes Deployment. *kubernetes.io.* [Online] Kubernetes. https://kubernetes.io/docs/concepts/workloads/controllers/deployment/.

16. Kubernetes StatefulSets. *kubernetes.io.* [Online] Kubernetes.
https://kubernetes.io/docs/concepts/workloads/controllers/statefulset/.
17. Kubernetes ReplicaSet. *kubernetes.io.* [Online] Kubernetes.
https://kubernetes.io/docs/concepts/workloads/controllers/replicaset/.
18. Kubernetes Secret. *kubernetes.io.* [Online] Kubernetes.
https://kubernetes.io/docs/concepts/configuration/secret/.
19. Kubernetes ConfigMap. *kubernetes.io.* [Online] Kubernetes.
https://kubernetes.io/docs/concepts/configuration/configmap/.
20. GraphQL. *graphql.org.* [Online] GraphQL. https://graphql.org/.
21. Does GraphQL replace REST? *graphql.org.* [Online] GraphQL.
https://graphql.org/faq/#does-graphql-replace-rest.
22. What is Knative. *redhat.com.* [Online] Redhat.
https://www.redhat.com/en/topics/microservices/what-is-knative.
23. Keep Digital Repository Library. *hcmiu.edu.vn.* [Online] http://keep.hcmiu.edu.vn:8080/.
24. MIT Dspace. *mit.edu.* [Online] https://dspace.mit.edu/handle/1721.1/7582.
25. Theses and state final exams. *vut.cz.* [Online] https://www.fit.vut.cz/study/theses/.en.
26. Samuel, Stuchlý. *Setup of Application-Computation On-Premise Mini-Cloud Based on Kubernetes.* Brno, CZ : Brno University of Technology, Faculty of Information Technology, 2021.
27. About Node.js. *NodeJs.* [Online] https://nodejs.org/en/about/.
28. Kubernetes Client. *Github.* [Online] https://kubernetes-client.github.io/javascript/modules.html.
29. Here’s What We Learned About Page Speed. *Backlinko.* [Online]
https://backlinko.com/page-speed-stats.